Encapsulación del patrón Observer utilizando eventos explícitos en ECaesarJ

Encapsulation of the Observer pattern with explicit events in ECaesarJ

Gabriela C. Pantoja,[1]* Ulises Juárez-Martínez[1]

[1] División de Estudios de Posgrado e Investigación, Instituto Tecnológico de Orizaba.
Av. Instituto Tecnológico 852, col. Emiliano Zapata. Orizaba, Veracruz, México
* Correo-e: pantojaaraoz@acm.org

PALABRAS CLAVE:

patrones de diseño, Observer, ECaesarJ, máquinas de estado

RESUMEN

El desarrollo de componentes de software proporciona un alto nivel de reutilización además de que ayuda ampliamente en el mantenimiento de sistemas de software. Encapsulando en forma de componente de software el patrón de diseño Observer, se aumenta el grado de reutilización debido a que la reutilización de una solución del nivel de diseño pasa al nivel de implementación en forma directa. Las piezas ya construidas se utilizan en sistemas de software sin preocuparse de su construcción, el enfoque es únicamente en su utilización. Debido a que el patrón de diseño Observer está asociado al patrón arquitectónico MVC y dada su capacidad modular de notificación ante los eventos que un sistema genera, su encapsulación como componente tiene un fuerte impacto en la arquitectura del software. Los resultados de dicha implementación en el lenguaje ECaesarJ muestran no sólo una mejora significativa respecto a otras implementaciones, también mejora la administración de eventos gracias al concepto de máquinas de estado.

KEYWORDS:

design patterns, Observer pattern, ECaesarJ language, state machine

ABSTRACT

Software components development provides high-level reuse and a better support for software maintenance. Encapsulating Observer pattern as a software component increases reuse degree because of a design solution has a direct implementation. Components already constructed are used immediately by software systems, and the approach is based only on its use. Since Observer is associated with MVC architectural pattern and its modular capacity of event notification, the encapsulation of this pattern has a strong influence on software architecture. The results show a significant improvement with regard to other implementations; also it improves event administration due to the concept of state machines.

Progmat,2014, 6,1; http://www.progmat.uaem.mx http://dx.doi.org/10.30973/progmat/2014.6.1/5

1 INTRODUCCIÓN

La reutilización en la ingeniería de software es considerada como un atributo de calidad en los productos de software debido a que ofrece beneficios en cuestiones de tiempo, costos y esfuerzo. La identificación de problemas recurrentes y sus soluciones son un ejemplo de reutilización en cuanto al diseño, dichos problemas se denominan patrones y han sido catalogados por Gamma [1] presentando nombre, objetivo, estructura, participantes y consecuencias. Sin embargo en este contexto la reutilización de problemas ya reconocidos muchas veces sólo se realiza en el nivel de diseño.

En cuanto a reutilización en el nivel de código existen componentes que proporcionan partes del mismo que se utilizan en sistemas de software de acuerdo con las necesidades que se presenten. Además los componentes proporcionan beneficios desde la construcción del software debido a que proveen piezas ya fabricadas y listas para su utilización y mejoran el mantenimiento y evolución de software debido a que los componentes, al ser piezas encapsuladas, permiten aislar la información y realizar modificaciones sin afectar el software que lo utilice.

Actualmente los lenguajes orientados a aspectos ofrecen un alto nivel de modularidad, lo cual permite formar componentes. Entre estos lenguajes destaca ECaesarJ, el cual está orientado a aspectos basados en eventos que facilitan la modularidad y el desarrollo de componentes reutilizables [2, 3]. El desarrollo de componentes impacta ampliamente el aspecto arquitectónico en la construcción de sistemas, lo cual se ve reflejado esencialmente en la reutilización. ECaesarJ es un lenguaje enfocado en la construcción de las líneas de productos de software. Éstas proponen un desarrollo ingenieril en el cual todos los productos están estandarizados y resuelven diferentes necesidades dentro del mismo contexto [4].

El desarrollo de componentes de patrones de diseño catalogados por Gamma es un tema en el que se reportan pocas implementaciones y la mayoría está en lenguajes orientados a objetos, lo cual no explota las características de modularidad de los lenguajes orientados a aspectos debido a su nivel de abstracción. En este articulo se presenta la encapsulación del patrón Observer debido a la importancia que tiene desde el punto de vista arquitectónico, pues forma parte del patrón MVC y por su característica de realizar una acción al cambio de estado. Los eventos y cambios de estado se ven beneficiados con el uso de eventos que ECaesarJ proporciona junto con una mejor manera de administrar dichos cambios. A su vez se destaca la reutilización del patrón como componente de software en la construcción de sistemas.

En la sección 2 de este artículo se presentan los trabajos relacionados; en la sección 3, el fundamento de la encapsulación del patrón Observer; en la sección 4 se discute sobre la implementación realizada en ECaesarJ, y finalmente en la sección 5 se muestran los resultados de la encapsulación en ECaesarJ.

2 ENCAPSULACIÓN DE PATRONES DE DISEÑO

Actualmente existen trabajos que reportan la implementación de patrones de diseño como una solución a la falta de reutilización a nivel de código, estas implementaciones se catalogan de acuerdo con el paradigma utilizado y el nivel de abstracción para la implementación de patrones.

2.1 Implementaciones de patrones de diseño como componentes

De acuerdo con Arnout [5], se realizó el análisis de los patrones catalogados por Gamma y se determinó que 15 de los 23 son factibles de encapsular. El trabajo de implementación se realizó con el lenguaje orientado a objetos Eiffel.

Según García [6], se realizó la encapsulación de los patrones de diseño de GoF en el lenguaje Java, tomando como base el trabajo reportado por Arnout [5], incluyendo también otros patrones relevantes, como una implementación genérica de *data access object*, además contiene una extensión para el patrón Observer incluyendo una implementación del patrón Proxy para proteger a los observadores.

2.2 Implementación de patrones de diseño en lenguajes orientados a aspectos

Si bien la encapsulación de patrones de diseño es deseable sobre cualquier mejora de implementación, no es posible descartar dichas mejoras con la utilización de lenguajes que integran o combinan otros paradigmas, tales como la orientación a aspectos y la programación funcional.

De acuerdo con Hannemann y colaboradores [7], se compararon las implementaciones de los patrones de GoF en el lenguaje Java y AspectJ y observaron que 17 implementaciones en AspectJ tuvieron mejoras en su estructura y capacidad de volver módulos las implementaciones realizadas.

Sousa y Monteiro [8] implementaron siete patrones de diseño en diferentes lenguajes orientados a objetos como lo son AspectJ y CaesarJ con el propósito de obtener un repositorio de implementaciones en lenguajes orientados a aspectos como las ya existentes en lenguajes orientados a objetos.

Por su parte, Oliveira y colaboradores [9] implementaron el patrón Visitor en el lenguaje Scala y garantizaron que se realizara de manera modular debido a características tales como la parametrización de tipos y tipos abstractos que Scala proporciona. Obtuvieron una biblioteca genérica que es integrable a sistemas para su reutilización.

3 ESTRUCTURA E IMPORTANCIA DEL PATRÓN OBSERVER

El patrón Observer, también conocido como publicación-suscriptor, notifica a los objetos que dependen de él si un objeto cambia de estado, de esta forma, los objetos se actualizan automáticamente de acuerdo con los cambios realizados [5, 10]. Este patrón se compone de dos objetos principales que son el sujeto y el observador. En la figura 1 se muestra la estructura del patrón Observer.

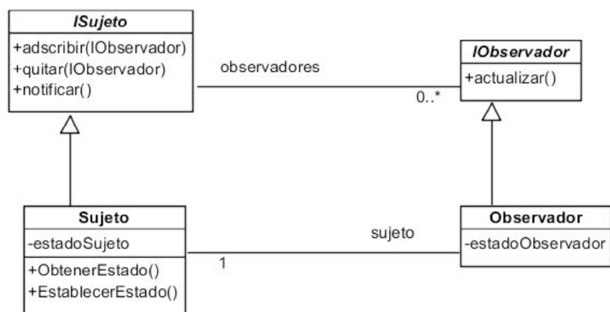

Figura 1. Estructura del patrón Observer

En este patrón el sujeto puede tener varios observadores que dependan de él y, cada vez que el sujeto cambie de estado, notifica a todos y cada uno de sus observadores del cambio que ha sufrido, a su vez los observadores actualizarán el estado del sujeto al cual están observando y ha indicado cambios, para obtener el nuevo estado. Este funcionamiento se muestra en el siguiente diagrama.

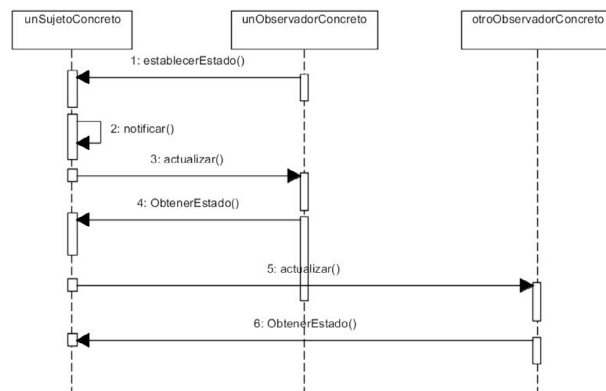

Figura 2. Secuencia del patrón Observer

3.1 Importancia arquitectónica del patrón Observer

Este patrón es uno de los más conocidos y al implementarlo en forma de componente se obtienen beneficios importantes. Es necesario tener clara la idea de lo que significa un componente: una unidad de composición con interfaces especificadas contractualmente. Estas interfaces establecen condiciones entre el proveedor del servicio y el usuario y son el punto de acceso de los usuarios para acceder al servicio que se presta. Estas condiciones son llamadas contratos, los cuales son especificaciones adjuntas a las interfaces que sirven como estándar para la utilización de componentes [11].

A diferencia de un objeto, un componente encapsula su estado y no es visible externamente, además puede componerse de un conjunto de clases y objetos y el componente no necesita conocer las referencias de los objetos que lo componen [11].

Desde el punto de vista arquitectónico el conjunto de componentes conectados entre sí forman una arquitectura, la cual permite definir la estructura de un sistema de software [12]. Cada uno de los componentes ofrece funcionalidad que ayuda a cumplir el objetivo por el cual se decide crear el sistema. La arquitectura determina la independencia que tendrá cada componente en la propia arquitectura para solucionar las funciones que tiene a su cargo y también determina la cooperación que tendrán los componentes entre sí, por lo tanto es importante que cada uno cuente con

los requisitos que la arquitectura propone, ya que son la parte medular de las arquitecturas de software.

Por lo tanto al encapsular el patrón Observer en forma de componente de software se ofrecen beneficios de reutilización arquitectónica, ya que es capaz de formar parte de diferentes arquitecturas en donde se identifique que es necesaria la utilización de dicho patrón.

El patrón Observer se incorpora en el conocido patrón arquitectónico modelo-vista-controlador (*model-view-controller*) (MVC), que propone la separación de una implementación. Ésta ayuda al mantenimiento de un sistema al estar definidos claramente los componentes que son parte de la vista, del control y del modelo. Con el patrón Observer es posible definir la interacción entre el modelo y la vista, lo cual se muestra en la figura 3. Debido a que la conexión entre el modelo y la vista es de manera dinámica y se produce a tiempo de ejecución, es necesario reflejar los cambios que se produzcan para mantener un estado de consistencia.

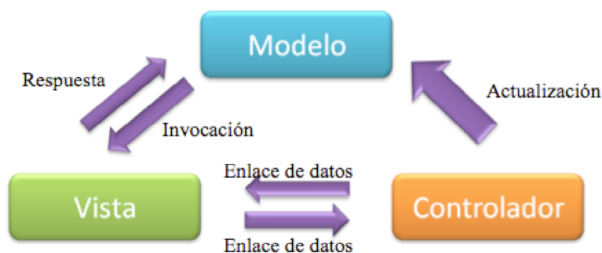

Figura 3. Patrón Observer en el patrón arquitectónico MVC

4 IMPLEMENTACIÓN DEL PATRÓN OBSERVER EN ECAESARJ

Para la encapsulación de este patrón de diseño se utilizó el lenguaje ECaesarJ, que integra características de su antecesor CaesarJ y el paradigma orientado a aspectos dirigido por eventos. ECaesarJ se enfoca en el desarrollo de líneas de productos de software debido a la capacidad de modularizar la variabilidad de una línea de productos de software mediante características como son clases virtuales, eventos y máquinas de estados.

Además, permite la identificación de eventos, es decir sucesos de interés para los observadores, los eventos que se permiten en este lenguaje son implícitos y explícitos. Los eventos explícitos, como su nombre lo dice, se especifican explícitamente por el desarrollador (se generan desde un origen), y sirven para que los destinatarios reaccionen según el evento que ocurrió en la fuente origen. Por otro lado, los de tipo implícito son comparados con elementos que expongan otros eventos o valores de campos o variables que estén definidas y a las que les sea posible acceder, es decir, cualquier acceso a variables, campos o ejecuciones de métodos y constructores. Además es posible utilizar operadores lógicos como lo son: && (and), || (or), y ! (not). Estos eventos se consideran implícitos porque no requieren notación adicional (explícita) por parte del desarrollador [1, 12].

Adicionalmente ECaesarJ implementa el mecanismo de máquinas de estado, el cual esencialmente es un manejador de eventos. Este mecanismo permite controlar de manera lógica los estados de un objeto en forma organizada, mejorando la flexibilidad y estabilidad de características basadas en la descomposición de comportamiento (no está disponible en lenguajes orientados a objetos como Java). Las máquinas de estados en ECaesarJ son definidas dentro de clases definiendo el estado inicial y las transiciones a cada estado.

Las características presentes en ECaesarJ para la implementación del patrón Observer impactan de manera importante debido a que ECaesarJ por naturaleza ofrece el soporte de eventos explícitos.

Para la encapsulación del patrón Observer es necesario identificar las partes importantes de este patrón, que son el sujeto y los observadores, como ya se mencionó.

El sujeto contiene los métodos “agregar”, “eliminar” y “notificar” al observador tal y como se muestra en las siguientes líneas:

```
1 public abstract cclassIObservable  {
2 public abstract voids
3          agregarObservador(IObservador
observador);
4 public abstract void
5          quitarObservador(IObservador
observador);
6    public   abstract   int   numeroOb-
servadores();
7 public abstract event no-tificar(Object
datos);
```

La notificación a los observadores corresponde a un evento, el cual se muestra en el siguiente código:

```
1    event notificar(Object datos);
2
3    notificar (Object datos)=>{
4    estado.setEstado(datos);
5    for(IObservador obs : observadores){
6    obs.actualizar();
7    }
8    }
```

La especificación del evento mostrado de las líneas 3 a la 8 tiene como argumentos el cambio de estado, que es de interés de los observadores y al ser invocado ese evento dispara el correspondiente a la actualización del estado por parte de los observadores.

De igual manera, la actualización que realizan éstos corresponde a un evento que se habilita una vez que ha identificado un cambio, el cual se muestra a continuación:

```
1    event actualizar();
2
3    actualizar ()=>{
4    estado = estadoSujeto.getEstado();
5    }
```

Estos eventos localizados corresponden a cambios que surgen directamente a un estado, sin embargo también existen eventos como la suscripción de un observador o su eliminación que se reflejan en el funcionamiento del patrón Observer y que son reflejados en esta implementación en la máquina de estados. Estos eventos permiten controlar y entender de mejor manera el funcionamiento del patrón en forma clara y con la secuencia lógica que en este patrón se observa.

En el siguiente código se muestra la especificación de los eventos que se proporcionan para el funcionamiento de los sujetos que requieren el uso de diversos estados:

```
1 state no_started=(
2 estado = "Not started";
3 );
4 state started=(
5 estado = "Started";
6 );
7 state suspended=(
8 estado = "Suspended";
9 );
10 state in_progress=(
11 estado = "In progress";
12 );
13 state completed=(
14 estado = "Completed";
15 );
16 state failed=(
17 estado = "Failed";
18 );
```

Estos estados permiten a los usuarios definir los cambios de estado que sufre cierto sujeto o dato a lo largo de su aplicación, para así tener un mejor control de los cambios, además de incrementar la flexibilidad en sus aplicaciones, gracias a las características que posee ECaesarJ.

Con esta implementación se logra aislar el funcionamiento del patrón que, en caso de ser utilizado en el patrón arquitectónico MVC, el sujeto formaría parte del modelo que cambia y notifica al observador, y ésta a su vez se reflejará en la vista, de acuerdo con los cambios que surgieron.

Al obtener esta encapsulación es posible incluir estas clases en sistemas sin necesidad de tener el mayor conocimiento del patrón, sólo basta con saber qué clases desempeñarán cada rol.

5 RESULTADOS OBTENIDOS

El resultado de esta encapsulación es una interfaz que ofrece los servicios que el patrón Observer proporciona. Dicha encapsulación puede ser utilizada solamente definiendo quiénes fungirán en los roles de sujeto y observador y cuál es el estado por observar.

Esta implementación es mucho más flexible y con un nivel de abstracción mayor y más semejante al patrón debido a la incorporación de eventos que son lanzados conforme un cambio aparece.

Además esta encapsulación permite su incorporación en forma transparente en la construcción de sistemas por ser un componente de software y con el beneficio adicional de trabajar en el nivel arquitectónico con el patrón MVC.

En comparación con la encapsulación realizada en Java, la encapsulación en ECaesarJ ofrece mayor abstracción debido a las características que brinda. Además de claridad, la característica de las máquinas de estados brinda más flexibilidad al poder incorporar nuevos estados utilizando las propiedades de los lenguajes orientados a aspectos sin alterar el funcionamiento básico del patrón Observer.

En la tabla 1 se muestra una comparativa de las implementaciones realizadas en Java y Eiffel [5, 6], de las cuales se pueden observar las diferencias significativas entre ellas y la implementación presentada.

Tabla 1. Comparación de implementaciones del patrón Observer

	Eiffel	Java	ECaesarJ
Flexibilidad	Sí	No	Sí
Escalabilidad	Sí	No	Sí
Claridad	Sí	Sí	Sí
Dispersión de código	No	Sí	No
Manejo de errores	No	No	Sí
Grado de abstracción	Medio	Medio	Alto
Evento explícito	No	No	Sí
Máquina de estado	No	No	Sí
Mantenimiento	Complicado	Complicado	Sencillo

Las características mostradas en el componente implementado proporcionan beneficios como la flexibilidad y escalabilidad en el desarrollo de sistemas, lo cual impacta esencialmente en el mantenimiento de sistemas de software. Por otro lado la claridad de la implementación concentrada en un mismo lugar permite que los programadores inexpertos sean capaces de utilizarla sin requerir un amplio conocimiento.

6 CONCLUSIONES Y TRABAJO A FUTURO

Los patrones de diseño permiten reutilizar diseños de soluciones a problemas frecuentes. Contar con reutilizaciones en forma de componentes simplifica el desarrollo de software, no sólo desde la etapa de diseño, sino además en las de mantenimiento. Por otro lado, desde el punto de vista arquitectónico, un componente aporta rapidez y consistencia y mejora la calidad en el desarrollo de software, también los componentes dentro de un sistema mejoran la evolución de los sistemas de manera más simple y estandarizada.

Finalmente es importante destacar la importancia en utilizar eventos y máquinas de estados. Ambos son mecanismos que facilitan la administración de eventos, permiten mayor flexibilidad en obtener modularidad no sólo como componentes estructurales, sino como componentes capaces de modularizar el comportamiento. Por supuesto que estas capacidades son altamente deseables en la implementación de otros patrones de diseño y patrones arquitectónicos.

Como trabajo a futuro se analizarán los patrones catalogados por Gamma y los orientados a aspectos en los cuales se observa el manejo de eventos para su posterior implementación en componentes de software en ECaesarJ. Dado el potencial de sustitución que ofrece un componente de software se evaluará la capacidad de sustitución entre patrones.

REFERENCIAS

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995). Design patterns: Elements of reusable object-oriented software. Addison Wesley.
2. Núñez, A., Noyé, J., Gasiunas, V. (2009). Declarative definition of contexts with polymor-phic events. En International Workshop on Context-Oriented Programming (COP '09).
3. Núñez, A., Gasiunas, V. (2012). ECaesarJ User's Guide http://ample.holos.pt/gest_cnt_upload/editor/File/public/ECaesarJ-manual.pdf
4. Pohl, K., Günter, B., Linden, F. (1998). *Software product line engineering foundations, principles, and techniques*. Springer.
5. Arnout, K. (2004). From patterns to components. Tesis de doctorado. Swiss Federal Institute of Technology.
6. Garcia, G. A. (2012). Perfectjpattern. http://perfectjpattern.sourceforge.net/
7. Hannemann, J., Kiczales, G. (2002). Design-pattern implementation in java and aspectj. En: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.
8. Sousa, E., Monteiro, M. P. (2008). Implementing design patterns in CaesarJ: an exploratory study. En: Proceedings of the 2008 AOSD workshop on Software engineering properties of languages and aspect technologies.
9. Oliveira, B. C., Wang, M., Gibbons, J. (2008). The visitor pattern as a reusable, generic, type-safe component. En Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications (OOPSLA).
10. Metsker, S. J. (2002). *Design pattern Java Workbook*. Addison Wesley.
11. Spersky, C., Dominik, G., Stephan, M. (2002). Component software, 2a ed. AddisonWesley.
12. Bass, L., Clements, P., Kazman, R. (2003). *Software architecture in practice*. Addison Wesley.

Acerca de los autores

Gabriela Cointa Pantoja Aráoz obtuvo el grado en Ingeniería en Sistemas Computacionales por el Instituto Tecnológico de Orizaba. Actualmente es estudiante de la Maestría en Sistemas Computacionales del mismo instituto y sus principales áreas de interés son la ingeniería de software y el desarrollo de software orientado a aspectos.

Ulises Juárez Martínez obtuvo el grado de Doctor en Ciencias en la especialidad de Computación por parte del Cinvestav-IPN en 2008. Su trabajo doctoral consistió en el de-sarrollo de un marco de trabajo para la pro-gramación orientada a aspectos de grano fino. Sus intereses incluyen el desarrollo de software orientado a aspectos, arquitecturas de software, líneas de productos de software, compiladores y TRIZ aplicado al software. Actualmente es profesor investigador en el Grupo de Ingeniería de Software del Instituto Tecnológico de Orizaba, Veracruz, México.