Ingeniería 22 (2): 45-60, ISSN: 1409-2441; 2012. San José, Costa Rica

USO DE PATRONES DE DISEÑO DE SOFTWARE: UN CASO PRÁCTICO

Isaac Montenegro Jiménez
Luis Rodríguez Rodríguez
Gabriela SalazarBermúdez

Resumen

Los patrones de diseño brindan soluciones a una serie de problemas comunes que se presentan en el desarrollo de software. Algunas soluciones son: facilitan la reutilización y la capacidad de expansión del software, reducen la complejidad del código y del acoplamiento, y facilitan el mantenimiento. Sin embargo, estas ventajas solo son posibles si el software es diseñado cuidadosamente. En este artículo se ejemplifican las bondades expuestas anteriormente y se explican los beneficios potenciales de cada patrón de diseño, a través de la aplicación de patrones de diseño en un proyecto de software de un curso de Pregrado en la Universidad de Costa Rica. El proyecto de software utilizado trata sobre un simulador de un procesador multinúcleo, el cual inicialmente posee restricciones que lo hacen muy simple, sin embargo, al aplicar patrones de diseño se puede extender la capacidad del procesador para simular una mayor diversidad de arquitecturas. El artículo va dirigido a profesionales o estudiantes de computación con conocimientos básicos en programación orientada a objetos y arquitectura de computadoras.

Palabras clave: Ingeniería de software, patrones de diseño, desarrollo de software, arquitectura de computadores.

Abstract

Among the major advantages of object-oriented programming is the ability to reuse previously created functionalities to increase application development productivity and improving or extending existing applications, without the need to re-work any unnecessary steps. Several additional benefits include better maintainability, reduced code complexity and lower coupling between application components. Nevertheless, these advantages are only possible if the software project has been carefully designed. Design patterns provide guaranteed solutions to a series of common problems, eases reuse and software expandability. In this article, these benefits are explan, analyzed and explained by applying design patterns in a software project in an undergraduate course of the Universidad de Costa Rica. The project used to demonstrate this is a multicore processor unit that initially has many restrictions, thus making it a very simple model; however, through the use of design patterns it is shown how they can extend the application to simulate a wider variety of processor architectures. This article is intended for Software Engineers-in-training with basic knowledge of Object Oriented design and Computer Architecture.

Keywords: Software engineering, design patterns, software development, computer architecture.

Recibido: 3 de agosto del 2011 • **Aprobado:** 2 de octubre del 2012

1. INTRODUCCIÓN

Los patrones de diseño proveen soluciones comprobadas a problemas comunes y por ende son un mecanismo poderoso para agilizar el desarrollo de software. Por esta razón, de acuerdo con Gómez, M., Jiménez, G., Arroyo, J. (2009) el conocimiento de patrones de software debería ser parte de la educación básica de los ingenieros y científicos de la computación. Idealmente deberían ser enseñados como métodos naturales de programación orientada a objetos desde los primeros cursos de la carrera según Gestwicki P., Sun F. (2008).

En el curso de Ingeniería de Software del Programa de Pregrado de la Escuela de Ciencias

de la Computación e Informática, uno de los temas tratados es el de patrones de diseño de software. En este curso los estudiantes deben desarrollar una aplicación web en la cual apliquen los patrones vistos en clase. Sin embargo, debido a la naturaleza de los proyectos del curso, sólo era posible aplicar algunos patrones, por lo que se decidió aprovechar el proyecto de otro curso con el fin de utilizar una mayor cantidad.

El experimento realizado se aplicó a un proyecto del curso de Arquitectura de Computadoras. El objetivo del proyecto es realizar la simulación de una computadora funcional, incluyendo sus partes principales: Unidad de Procesamiento Central (CPU, por sus siglas en inglés), memoria de acceso aleatorio (RAM, por sus siglas en inglés), y el bus de comunicaciones entre estos dos. Se simularon varios CPU basados en la arquitectura MIPS (*Microprocessor without Interlocked Pipeline Stages*, por sus siglas en inglés), con sus cachés de datos e instrucciones, así como el código y los datos almacenados en la memoria RAM para el programa virtual que se simula.

En la sección 2, se presenta el marco teórico con los conceptos básicos utilizados en el artículo. En la sección 3, se describe el proyecto que se utilizó como base para realizar el experimento. En la sección 4, se describe la metodología seguida para seleccionar y aplicar cada patrón, en la sección 5, se analizan los resultados obtenidos. Posteriormente, en la sección 6, se expone el trabajo que se puede realizar a futuro y finalmente, en la sección 7, se presentan las conclusiones obtenidas del experimento.

2. MARCO TEÓRICO

En esta sección se describen los conceptos básicos utilizados en el artículo.

2.1 Patrones de diseño

En el ámbito de la computación, según Larman (2004), un patrón de diseño es una descripción de un problema y su solución, a la cual se le da un nombre, y se puede aplicar a nuevos conceptos. Es decir, un patrón provee una solución aceptada a un problema común y una terminología para distinguir esa solución. A continuación se describen brevemente los patrones utilizados en la investigación con base en Larman (2004).

2.2.1 Polimorfismo

Cuando existen comportamientos que varían según el tipo de objeto y se desea que el diseño sea extensible, se utiliza el patrón Polimorfismo. Básicamente se define un nombre para un servicio o comportamiento y se implementan diferentes versiones de ese servicio o comportamiento en diferentes objetos. La ventaja de este patrón radica en evitar dependencias lógicas condicionales como cadenas de sentencias *if-else* o *switch*, las cuales son poco extensibles.

Un ejemplo de este patrón puede ser una aplicación de alarma para un dispositivo móvil. El usuario puede elegir entre reproducir un sonido o hacer que el dispositivo vibre una vez se active la alarma. En lugar de crear un solo tipo de objeto alarma que condicionalmente active el vibrador o reproduzca un sonido cuando se active la alarma, se pueden definir dos tipos de objeto alarma, ambos con un método *activarAlarma* y cada objeto tendrá una implementación distinta de dicho método. De esta forma, se torna sencillo crear tipos nuevos de alarmas con funcionalidad variada, como por ejemplo encender la pantalla o activar la radio en una emisora predeterminada.

2.1.2 Fábrica (o Factoría)

En muchas circunstancias, la creación de objetos requiere lógica compleja o condicional. En estos casos es deseable separar esta responsabilidad para mejorar la cohesión entre los objetos y abstraer la lógica de creación. Para lograr este objetivo es apropiado utilizar el patrón Fábrica, el cual consiste en crear un objeto que encapsule y se encargue de la creación de objetos.

Para ejemplificar este patrón, se puede considerar el desarrollo de un reproductor multimedia que debe soportar distintos formatos de audio

y video. Para cada tipo de formato existe una implementación distinta que posee la lógica para decodificar y reproducir cada formato. En este caso se crea un objeto Fábrica que, dependiendo del tipo de archivo a reproducir, cree el objeto correspondiente. De esta forma, para soportar un nuevo formato, sólo es necesario implementar el nuevo tipo de objeto que lo maneje y modificar la Fábrica para que reconozca este nuevo tipo de formato y cree el nuevo objeto. El resto del código permanece intacto.

2.1.3 Singleton

Cuando existe un objeto del cual debe existir únicamente una instancia y es deseable un acceso global a esta instancia se utiliza el patrón Singleton. Comúnmente un Singleton se implementa definiendo un tipo de objeto que posee como uno de sus miembros una única instancia de sí mismo y un método estático llamado, por ejemplo, *getInstancia*, el cual retorna una referencia a esta única instancia del objeto. Cuando en alguna parte del código se requiera utilizar un servicio del Singleton, se llamará primero al método *getInstancia* para obtener la instancia global del objeto y de esta forma, a los servicios del objeto por medio de esta instancia.

Un ejemplo de su uso es el acceso al componente del sistema de posicionamiento global (GPS) de un dispositivo móvil. Debido a que existe sólo un componente que puede ser accedido desde varias partes de una aplicación, e incluso desde varias aplicaciones, es conveniente crear un Singleton para acceder a los servicios del componente. De esta forma desde cualquier parte de una aplicación se puede ejecutar un código como *ControladorGPS.getInstancia().getPosicionActual()* para obtener la posición actual. Esto provee el acceso global al controlador y permite mantener una sola instancia del controlador.

2.1.4 Indirección

Cuando se requiere agregar una responsabilidad que involucra dos componentes, pero que asignar esa responsabilidad a cualquiera de los dos componentes disminuiría su reusabilidad, se acude al patrón Indirección. Este patrón consiste en crear un objeto intermedio que maneje la comunicación entre los dos componentes.

Para ejemplificar este patrón, se puede considerar una aplicación que permita compartir contenido a través de varios medios, incluyendo redes sociales, correo electrónico, entre otros. Para cada medio existe un componente distinto con interfaces distintas. En este caso, en lugar de crear código que enfrente cada tipo de medio en cada parte de la aplicación donde se pueda compartir contenido, es mejor crear una Indirección que maneje estas conexiones. De esta forma cuando se desee compartir contenido, es posible hacerlo a través de la Indirección. Si eventualmente el componente para enviar correos electrónicos cambia, simplemente hay que modificar la Indirección y todas las partes de la aplicación que comparten contenido por correo electrónico permanecen intactas.

2.1.5 Observador

El patrón Observador se utiliza cuando un objeto genera un evento o realiza un cambio de estado y varios componentes deben reaccionar a éste de forma distinta. El patrón consiste en definir una interfaz que implementa todos los objetos que estén interesados en un evento. El objeto que genere el evento avisará, por medio de esta interfaz, a todos estos componentes que el evento ha ocurrido. Entonces cada componente reacciona de forma distinta, según como fue implementado.

Un ejemplo de uso se puede encontrar en un reproductor de música. Si el usuario está escuchando música y desconecta los audífonos, el sistema genera un evento que reciba tanto la interfaz, como el controlador del reproductor. La interfaz entonces muestra un mensaje advirtiendo que los audífonos fueron desconectados y el controlador pausa la canción o busca un medio de salida de audio alternativo para reproducir el audio.

2.2 Arquitectura de un computador

En esta sección se describen las partes básicas que contiene un computador, además se explica la arquitectura MIPS de un CPU.

2.2.1 Partes de un computador

John von Neumann (1945) propone que un sistema automático de computación, conocido actualmente como computador, debe tener varias partes principales para su funcionamiento. Entre ellas se tiene una unidad de cálculos lógicos y aritméticos, una unidad de control de programa, y una sección de memoria en donde se guarda toda la información de datos y programa. Todas estas están conectadas entre sí, hoy en día a esta conexión se le llama Bus. En la Figura 1, se presenta un diagrama general de la misma.

ALU - Unidad Aritmético Lógica

Para realizar las operaciones necesarias para el funcionamiento del trabajo que se le solicita al computador, se necesita una unidad que pueda recibir datos e instrucciones y producir un resultado con éstos. A esta unidad se le llama ALU (Arithmetic-Logic Unit) y tiene capacidad para realizar distintas operaciones lógicas y matemáticas, desde las operaciones básicas como sumas y restas, inclusive operaciones más complejas, como operaciones trigonométricas. El CPU se encarga de darle la información requerida al ALU y de recibir el resultado producido.

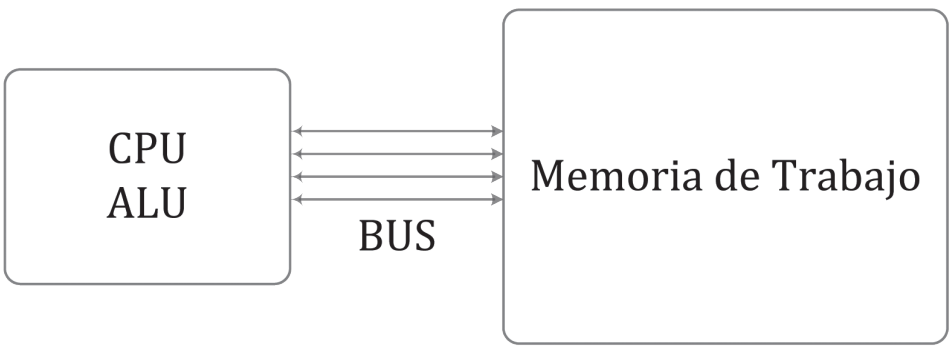

Figura 1. Arquitectura de Von Neumann.
Fuente: elaboración propia.

CPU - Unidad de Procesamiento Central

Según von Neumann (1945) el CPU (Central Processing Unit) o Unidad de Procesamiento Central es el "cerebro" que controla todas las funciones de la computadora. Como se desea que una computadora sea de propósito general, se tiene que tener un componente que controle el funcionamiento de todo el sistema.

Este componente se encarga de coordinar todas las acciones del sistema. Esto incluye enviar a ejecutar nuevas instrucciones, monitorear las tareas que se están realizando en un momento dado y administrar los recursos del sistema, especialmente el manejo de la memoria.

En las computadoras usadas actualmente, el ALU está incluido dentro del mismo chip que contiene el CPU, para lograr mayor velocidad de procesamiento. Un ejemplo es la arquitectura MIPS, que se discutirá posteriormente.

Memoria

Para realizar operaciones complejas y extensas en donde se tenga que guardar información conforme se realiza una operación, se necesita una memoria en donde colocar estos datos. Por ejemplo, al realizar una división, se tienen resultados parciales con los que hay que trabajar, éstos se almacenarán en

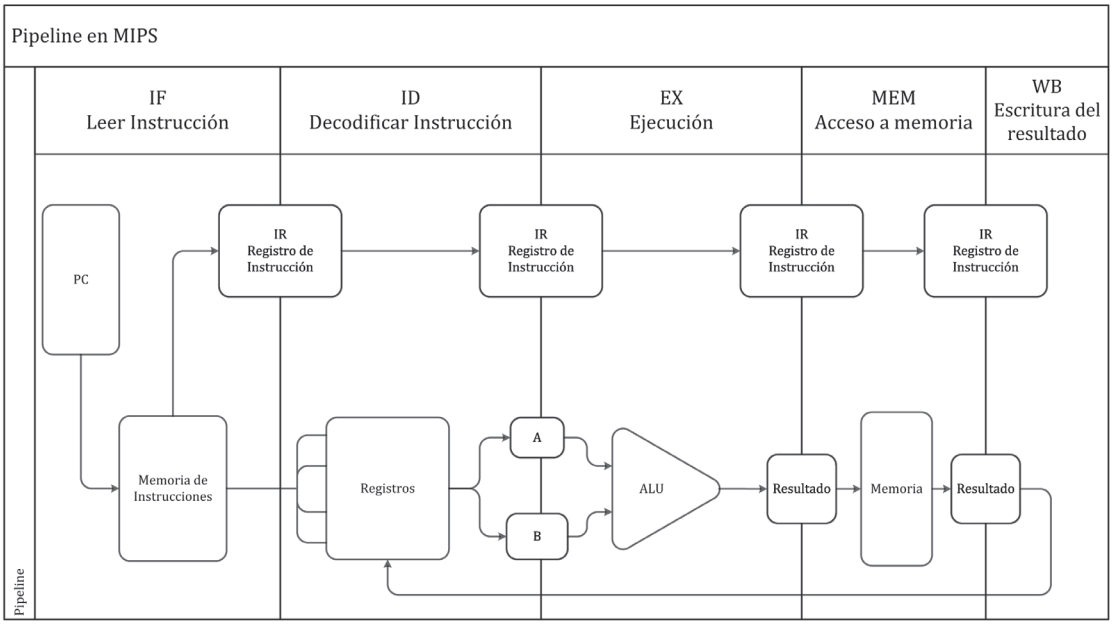

Figura 2. *Pipeline* simplificado en MIPS.
Fuente: elaboración propia.

la memoria. En la actualidad, a esta memoria se le llama memoria RAM (Random Access Memory) o memoria de acceso aleatorio. Esto se debe a que este tipo de memoria es implementada de modo que es posible acceder directamente a cualquier dato almacenado en ella, independientemente de su posición.

Bus

Para que todas las partes mencionadas anteriormente puedan trabajar en conjunto, se requiere de un Bus que las conecte entre sí. Además de conectar el CPU, ALU y Memoria, tiene como responsabilidad conectar con dispositivos que permitan leer y escribir programas a la computadora, así como leer los resultados de los programas ejecutados en la computadora.

El bus es requerido por todas las partes de la computadora, lo que lo convierte en un cuello de botella para todo el rendimiento del sistema. Una solución propuesta es darle al CPU una pequeña memoria que es copia de la sección de memoria que está trabajando en un determinado momento. A esto se le llama caché, y es utilizado en la simulación que se trabaja en este artículo.

2.2.2 Arquitectura MIPS

Patterson (2007) define la arquitectura MIPS (*Microprocessor without Interlocked Pipeline Stages*, por sus siglas en inglés) como un tipo de CPU que utiliza un set de instrucciones reducido (RISC, *Reduced Instruction Set Computer*).

RISC es un tipo de instrucciones de computadora sencillas de procesar, y tiene como propósito ejecutar muchas instrucciones rápidamente debido a su sencillez. Esto también permite ejecutar la siguiente instrucción antes de terminar de ejecutar la instrucción actual, lo que mejora el rendimiento. A esto se le llama *pipeline*. En la Figura 2, se muestra un diagrama simplificado del *pipeline* de la arquitectura MIPS.

Como se mencionó en la sección 2.2.1, los CPU modernos incluyen muchas funciones, incluyendo el ALU, en su proceso de control de la ejecución del sistema, o *pipeline*. El *pipeline* es la estructura interna del CPU que ejecuta las instrucciones que recibe.

Una analogía sobre el *pipeline* en la vida real es una línea de manufactura de productos. Por ejemplo, una fábrica de juguetes. En dicha fábrica se tiene una línea de producción con 5 estaciones, y una persona trabajando en cada estación en la creación de muñecos de acción. Se pretende que todos los operarios en las estaciones estén siempre ocupados.

En la primera estación, el operador toma un muñeco en blanco de la caja de muñecos y se lo pasa a la siguiente estación. Sin esperar a que el muñeco que acaba de pasar sea completado en la última etapa, se prepara para tomar el siguiente muñeco. La segunda estación recibe el muñeco y le dibuja la cara y lo pasa a la tercera. Ahora, al muñeco se le pone pelo en la tercera estación y es movido a la cuarta. En la cuarta estación, el operador le pone la ropa al muñeco. Una vez que la quinta estación recibe el muñeco, lo pone en una caja para terminarlo. Una vez el muñeco está terminado, es pasado a bodega, donde se reúnen muchos muñecos terminados para su distribución y venta.

Etapas del *pipeline*

En esta sección, se definen en breve las etapas del *pipeline* acorde a las definiciones de Patterson (2007). Cabe mencionar que en general, cada etapa del *pipeline* hace dos acciones, procesar los datos recibidos de la etapa anterior, y enviar el resultado a la siguiente etapa. A estas acciones se les denomina parte baja y parte alta del ciclo de ejecución.

- IF - Cargar Instrucción
  En esta etapa (IF, Instruction Fetch), se carga la instrucción siguiente del programa para alimentar al pipeline. Se tiene un contador llamado PC (Program Counter) o contador de programa, el cual le indica al CPU por dónde se está ejecutando el programa.

- ID - Decodificar Instrucción
  En esta etapa (ID, Instruction Decode) se decodifica la instrucción recibida para saber qué se debe realizar. Dentro de esta etapa, hay una pequeña memoria que utiliza el CPU con información relevante a la acción a realizar, por ejemplo los operandos de una suma. Estos son llamados registros.
  Al decodificar la instrucción en la parte baja del ciclo, se sabe qué se necesita realizar. Entonces, en la parte baja se escogen los registros a enviar a la siguiente etapa para su ejecución. Por ejemplo, se envían ambas partes de la suma y la instrucción de sumar.

- EX - Ejecución de la Instrucción
  En esta etapa (EX, Execute) se encuentra contenido el ALU. Esta recibe los registros y la instrucción enviadas en la etapa ID y realiza la operación. Envía los resultados a la siguiente etapa, MEM.

- MEM - Acceso a memoria
  En esta etapa (MEM, Memory Access) se realiza el acceso a la memoria del sistema, ya sea para leer o escribir información. La acción en esta etapa depende de la instrucción recibida y del resultado obtenido del ALU en la etapa EX, la anterior. Si hay una instrucción de guardar o traer algo de memoria, es realizado en esta etapa del pipeline.

- WB - Escritura del resultado
  La última etapa (WB, Write Back) es la de escritura del resultado de la instrucción. En esta etapa se escribe el resultado de la operación a los registros del CPU. El resultado puede indicar el resultado de la operación realizada, o nada más un indicador de que la operación fue realizada exitosamente.

3. DESCRIPCIÓN DEL SIMULADOR

En esta sección se describen los requerimientos de la aplicación a desarrollar y la estructura básica del diseño del programa.

3.1. Enunciado del proyecto

El proyecto elegido para experimentar la aplicación de patrones consiste en un simulador de

un procesador de 32 bits para el manejo de números enteros. El procesador a simular es multinúcleo y permite ejecutar 3 hilos al mismo tiempo.

El programa creado debe cumplir con ciertos requerimientos, los más importantes son descritos a continuación.

- Los núcleos deben tener la capacidad de trabajar de forma independiente y a su vez cada núcleo debe implementar paralelismo a nivel de instrucciones. Esto se logra por medio de un *pipeline* de 5 etapas, donde cada etapa ejecuta una parte de cada instrucción, de modo que hasta 5 instrucciones pueden estar ejecutándose a la vez. Las etapas usadas están modeladas según la arquitectura MIPS.
- Debe existir un sistema operativo que gestione la distribución de los hilos a los núcleos usando una estrategia de *round-robin*. Es decir, los hilos estarán almacenados en una cola. Cuando un procesador esté libre, se tomará el primer hilo de la cola y será ejecutado en ese núcleo por un número de ciclos predeterminado. Cuando este tiempo se acabe, el hilo será sacado del núcleo y colocado en el final de la cola.
- Todos los hilos deben compartir la misma memoria. Cada núcleo se comunica con la memoria por medio de dos cachés de memoria. Un caché para traer de memoria las instrucciones a ejecutar y un caché para cargar y guardar datos cuando una instrucción lo requiera. Todos los cachés de datos se comunican con la memoria principal por medio de un solo bus que puede ser accesado por solo una caché a la vez. Igualmente debe existir un bus para las cachés de instrucciones.
- Las cachés se deben comportar como las cachés de los procesadores reales y deben almacenar ciertos bloques de memoria de forma que si un hilo requiere accesar al mismo dato varias veces, la caché pueda suministrar el dato sin necesidad de esperar por el bus e ir a la memoria principal. Se debe asegurar que cuando una caché haga un cambio en un bloque de memoria que esté almacenado en otras cachés, estas cachés invaliden sus datos locales. Es decir, si dos cachés poseen el mismo dato almacenado y una lo modifica, la otra invalidará este dato y cuando se lo pidan, lo volverá a traer de memoria y refrescará sus datos locales.
- Finalmente, debe existir una interfaz que muestre al usuario en todo momento el estado actual de cada núcleo, incluyendo el número de ciclos que ha ejecutado, el valor de todos sus registros internos y las instrucciones que está ejecutando.
- Como se puede ver en la Figura 3, el sistema tendrá múltiples procesadores (CPU) con sus respectivas cachés, memoria de trabajo (RAM) que contiene datos e instrucciones de programa y un bus que permite la comunicación entre ambas partes. Hay una interfaz que lee los datos de todo el sistema en vivo, mientras trabaja.

3.2. Estructura del proyecto

Para facilitar su construcción el proyecto fue dividido en dos secciones: la lógica del procesador y la simulación de la memoria. Ambas secciones se desarrollaron paralelamente. El procesador fue construido para ser lo más general posible por medio de interfaces y desde ahí se implementó la versión MIPS del procesador que era la solicitada en el proyecto.

Como se puede observar en la Figura 4, el sistema se organizó de modo que cada núcleo de procesamiento no conozca la etapas de procesamiento en sí, sino que conozca un objeto Etapas que es el que maneja las etapas.

El núcleo es el componente que ejecuta los programas en el simulador y posee únicamente un método llamado correr, descrito mediante el siguiente pseudocódigo:

```
etapas.iniciar()
mientras todas las etapas sigan ejecutándose
si todas las etapas terminaron un ciclo
avanzar el ciclo
si el número de ciclos es mayor que el número
de ciclos máximo
señalar a las etapas que terminen de ejecutarse
etapas.avanzarCiclo()
etapas.terminarEjecucion()
```

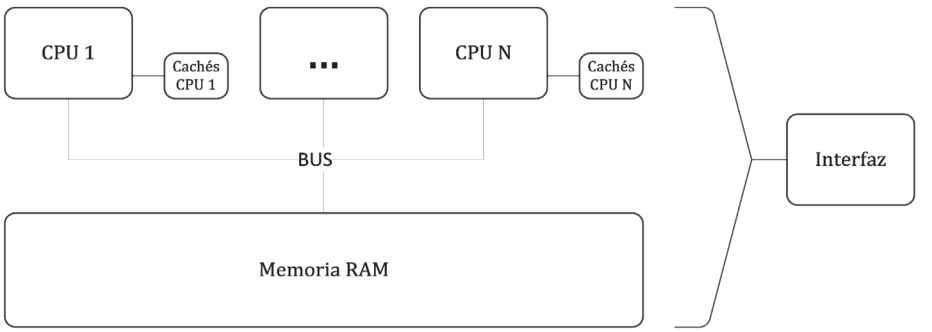

Figura 3. Diagrama general de la estructura a simular.
Fuente: elaboración propia.

De esta forma la implementación de las etapas no afecta la ejecución del núcleo y el objeto Etapas provee un comportamiento similar al de un adaptador.

En la siguiente sección se describe la forma en que los patrones de diseño influenciaron el diseño de la arquitectura de la aplicación.

4. METODOLOGÍA

El proyecto fue realizado de manera virtual, es decir, utilizando código para simular el sistema, y no hardware físico. Tomando esto en cuenta, se debe crear un diseño apropiado del código para que el proyecto sea fácilmente mantenible, extensible y legible ante otros diseñadores ajenos al proyecto.

Para cumplir este objetivo, se diseñó la aplicación teniendo en mente los distintos tipos de patrones, y se identificaron componentes y conexiones entre los mismos, donde era conveniente aplicarlos. Posteriormente se efectuó un análisis sobre la efectividad de haber aplicado estos patrones de diseño.

El formato aplicado en el estudio es del tipo exploratorio. El experimento está basado en la aplicación de diversos métodos implementados por educadores para mejorar la enseñanza de patrones mediante métodos prácticos. Entre los casos estudiados se encuentran la enseñanza de patrones mediante el análisis de problemas reales de acuerdo con Cinnéide M., Tynan R. (2004), el desarrollo de juegos de computadora según Gestwicki P., Sun F. (2008) y Gómez, M., et al. (2009) y el uso de proyectos altamente cambiantes con base en Ritzhaupt, A., Zucker R. (2009). En todos los casos se usaron proyectos prácticos como plataforma para enseñar patrones.

En las siguientes secciones se describen los patrones que fueron utilizados en el proyecto y un análisis de las ventajas obtenidas por cada uno de ellos.

4.1. Polimorfismo

Como se describió en la Sección 3.2, el diseño se inició con una estructura en donde las etapas están ocultas al núcleo. Para lograr esto, se definió una etapa abstracta que describe el comportamiento de cualquier etapa genérica. Esto se logra mediante el uso del patrón Polimorfismo. Las etapas se pueden implementar para adaptarse al tipo de arquitectura que se desee. El objeto polimórfico Etapa se implementó con acceso a los registros del núcleo y posee una única interfaz *correr*, descrita mediante el siguiente pseudocódigo:

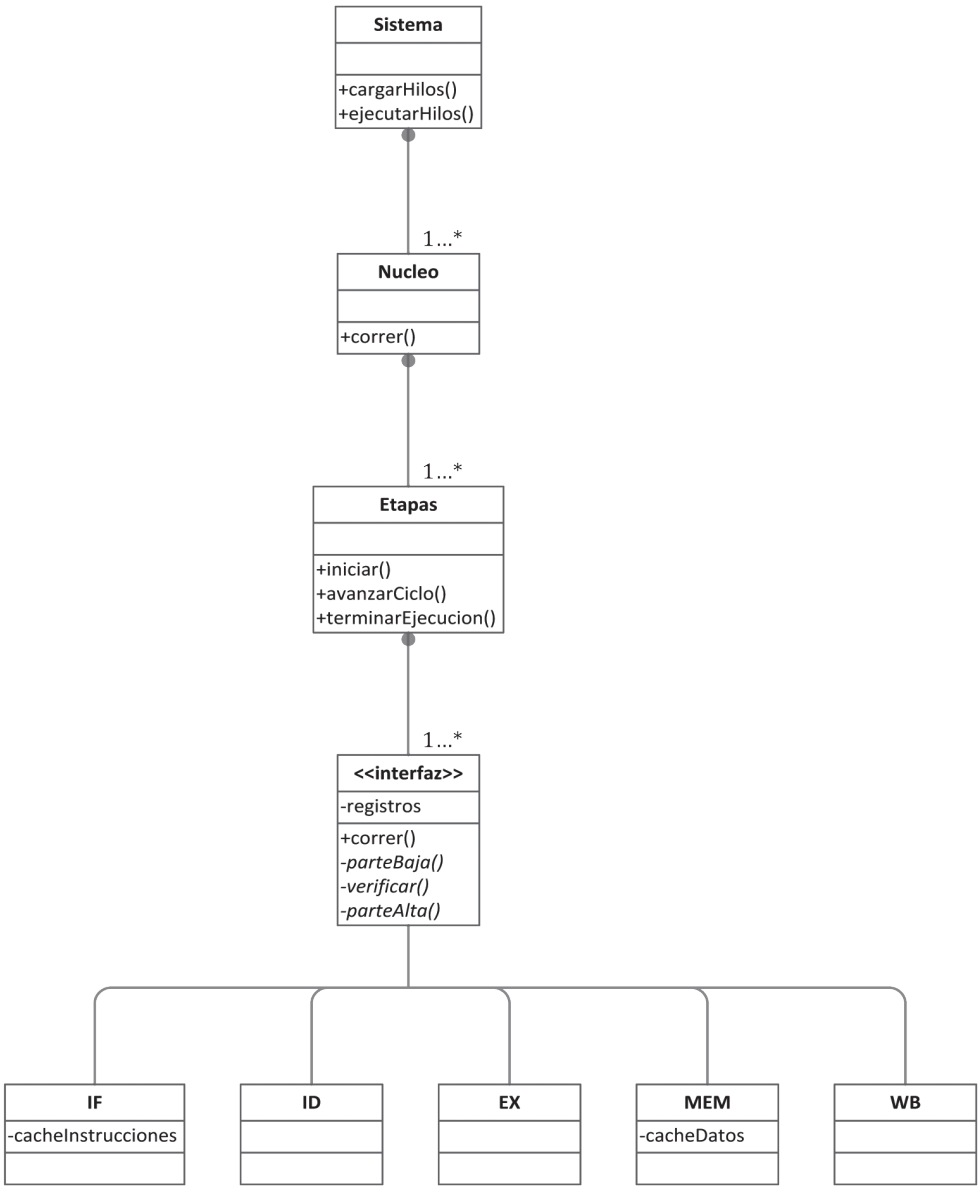

Figura 4. Descripción de la estructura del simulador del procesador sin considerar la sección de memoria.

Fuente: elaboración propia.

```
mientras no se haya señalado que termine
parteBaja()
esperar a que todas las etapas terminen su parte baja
verificar()
parteAlta()
esperar a que se señale que cambie de ciclo
```

Los tres métodos *parteBaja*, *verificar* y *parteAlta* son abstractos y son los únicos métodos que deben implementar las etapas según requiera la arquitectura. Por ejemplo, los requerimientos del proyecto exigen implementar la arquitectura MIPS, la cual consiste de cinco etapas. La mayoría de las etapas implementan el método *parteBaja* para cargar los datos que necesitan de los registros (los cuales son heredados de la Etapa genérica), usan su método *verificar* para comprobar que ninguna otra etapa está en un estado en el que evite que la etapa actual se ejecute y si el método de *verificar* no encontró problemas, usan el método *parteAlta* para ejecutar operaciones sobre los datos cargados y guardan sus resultados en los registros. Sin embargo, en la arquitectura MIPS, la última etapa es un caso especial, debido a que ésta guarda sus resultados antes que todas las demás etapas y no realiza otro trabajo posteriormente. Aún así, no es necesario crear código especializado para manejar este caso. Simplemente esta etapa utiliza el método *parteBaja* para cargar y almacenar datos y posee una implementación vacía en sus métodos *verificar* y *parteAlta*.

El uso del patrón Polimorfismo permite implementar diferentes arquitecturas con solo definir nuevas etapas y sin necesidad de cambiar ni la lógica del sistema, ni de los núcleos de procesamiento.

4.2. Fábrica y Singleton

Ahora que la implementación de las etapas fue separada de la lógica principal y se adaptó para poder soportar cualquier tipo de arquitectura, se tuvo que definir una estrategia para que el núcleo pueda definir sobre qué arquitectura está trabajando. Era necesario considerar que la definición de los registros depende de la arquitectura que se está utilizando, por lo que los registros también debían ser extensibles y adaptados con la arquitectura actual.

Debido a que el problema presente es separar la lógica de creación de objetos, la solución es aplicar el patrón Fábrica para definir un servicio que maneje la creación de objetos Etapas de distintas arquitecturas y de forma transparente para el sistema. Como se puede ver en la Figura 5, la FabricaDeEtapas posee métodos para crear tanto registros como etapas de la arquitectura MIPS. Una vez que se ha definido la arquitectura, cada vez que se ocupe crear objetos Etapas o Registros, se crearán por medio de la FabricaDeEtapas. La Fábrica conoce la lógica de creación de estos objetos y se encarga de las relaciones entre ellos. De esta forma esta lógica compleja es invisible para la implementación del Núcleo.

Con la FabricaDeEtapas, es posible simplificar mucho el proceso de implementar diferentes arquitecturas, debido a que después de crear la implementación de los registros y las etapas de la nueva arquitectura, basta con crear un método en la Fábrica que maneje la creación del conjunto de etapas y utilizar estos métodos para usar la nueva arquitectura, el resto del código queda intacto.

La Fábrica fue implementada utilizando el patrón Singleton, con el fin de obtener un acceso global a los servicios de la Fábrica, sin necesidad de crear instancias de la misma. Esta estrategia mejora el ocultamiento de la lógica de creación de objetos, simplifica el código y mejora la legibilidad del mismo.

4.3. Indirección

Aún después de aislar la lógica del sistema de la implementación de la arquitectura, se encontró un problema más que resolver, a saber: el núcleo de procesamiento debía ser capaz de comunicarse con las etapas de modo que les pueda avisar cuándo deben detenerse para poder realizar un cambio de contexto (sacar el hilo de procesamiento actual y reemplazarlo por un nuevo hilo). Además, las etapas deben poder comunicarse entre sí para que puedan saber cuándo pueden empezar a ejecutar su método *parteAlta* y para que sepan si una etapa posterior afecta su progreso. Por ejemplo, si una etapa debe esperar

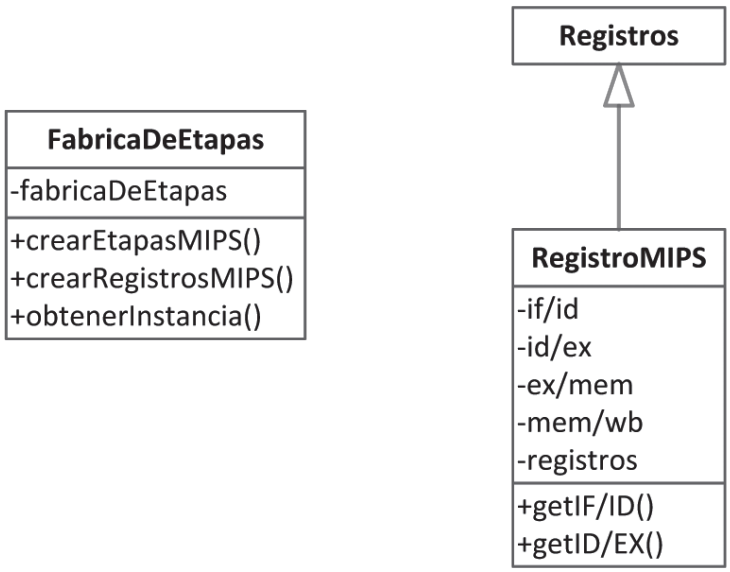

Figura 5. Definición de la Fábrica de etapas y los registros.
Fuente: Elaboración propia.

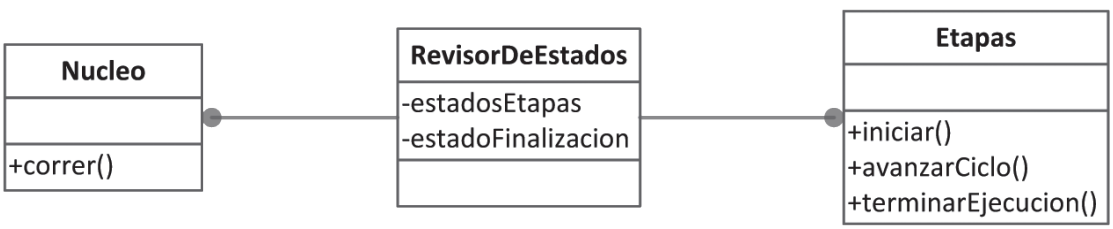

Figura 6. Comunicación entre el núcleo y las etapas.
Fuente: elaboración propia.

varios ciclos para acceder a un objeto en memoria, todas las etapas que están antes de ésta deben esperar también, de otro modo las etapas podrían empujar instrucciones hacia la etapa que está detenida y el sistema colapsa.

Para resolver este problema se decidió crear una Indirección entre el Núcleo y el objeto Etapas y otra Indirección entre todas las etapas. Como se puede ver en la Figura 6, tanto el Núcleo como el objeto de Etapas comparten un objeto llamado RevisorDeEstados. Cuando el Núcleo encuentra que se ha terminado el número de ciclos que le corresponden al hilo, señala a los hilos que deben terminar de ejecutarse por medio del RevisorDeEstados. De igual manera, para que el Núcleo sepa el momento en que todas las etapas han terminado de ejecutar un ciclo, las etapas indican al RevisorDeEstados cada vez que terminan de ejecutar un ciclo. El Núcleo revisa cada cierto tiempo el revisor de estados y cuando encuentra que todas las etapas terminaron, les avisa a todas que ejecuten el próximo ciclo.

La forma en que se comunican las etapas entre sí es muy similar. Si una etapa debe detener su ejecución por alguna razón, avisa a las demás etapas por medio de un objeto compartido.

4.4. Observador

Una vez que se ha definido la arquitectura del programa, falta un último problema a resolver a saber, la visualización de la información en una interfaz de usuario.

Los requerimientos solicitan que la interfaz de usuario debe desplegar la información de todos los núcleos a la vez, sin embargo, debido a que los núcleos se ejecutan de forma independiente, unos pueden ejecutarse más rápidamente que otros. Para solucionar el problema se usó el patrón Observador. Se creó un objeto que posee una referencia al estado de todos los hilos que están ejecutándose. Cada vez que el estado de un hilo cambia (por ejemplo, una etapa del núcleo cambia el estado de uno de sus registros), se lo señala a la interfaz, la cual refresca la pantalla y muestra la información actualizada. La interfaz no accede directamente a los detalles de implementación, sencillamente toma los datos del objeto Observador por lo que el acoplamiento entre el sistema y la interfaz es mínimo.

4.5. Otras consideraciones del diseño

Se puede notar que todos los patrones utilizados en el diseño del sistema aplican únicamente a la sección del procesador y no hubo mención a patrones utilizados en la administración de la memoria. Esto se debe a que la aplicación de patrones de diseño suele ser considerablemente más costosa de implementar que los métodos convencionales. Como la arquitectura de la memoria de las computadoras se ha mantenido constante por muchos años, los cambios que se podrían dar no suelen ser muy significativos por lo que se consideró poco provechoso implementar un sistema flexible en esta área. Sin embargo, no se descarta la posibilidad y se considerará más adelante en la sección de trabajo futuro.

Ahora que se ha explicado la forma en que el proyecto fue implementado utilizando patrones de diseño, se puede realizar una retrospectiva para analizar los beneficios y perjuicios producidos por los mismos, tanto en el producto final como en el proceso de desarrollo.

5. RESULTADOS

A continuación se analizan los resultados después de haber utilizado los patrones: Polimorfismo, Indirección, Factoría, Singleton y Observador.

5.1. Polimorfismo

Al utilizar Polimorfismo para definir genéricamente las etapas que son ejecutadas para cada instrucción, el sistema no queda restringido a simular únicamente un procesador MIPS. Es posible cambiar la implementación interna de las distintas etapas del núcleo y de los registros, sin afectar en sí el código que maneja la ejecución. De esta manera es sencillo modificar el programa para que simule una arquitectura distinta o una versión alterna de la misma arquitectura. De no haber utilizado Polimorfismo, cada vez que se quisiera experimentar con alguna otra arquitectura, sería necesario modificar muchas partes del código lo que reduce la mantenibilidad e incrementa los puntos de fallo.

El Polimorfismo hace que la aplicación sea más compleja de implementar para el desarrollador, y aunque permite tener futuras modificaciones de manera más sencilla, hay situaciones en las que no es recomendable aplicar el patrón. Por ejemplo, como se mencionó anteriormente, no es provechoso aplicar este patrón en el área de administración de la memoria, debido a que ésta no tiene variaciones significativas a lo largo del tiempo.

Además, como la aplicación tiene una mayor complejidad al utilizar el Patrón Polimorfismo, el proceso de pruebas se dificulta considerablemente.

5.2. Indirección

Utilizar el patrón Indirección facilita la creación de un objeto que pueda contener las diferentes variaciones de objetos, creadas previamente con el Polimorfismo, como se puede apreciar en la Figura 6 con el AdaptadorEtapas y el RevisorDeEtapas. De esta manera, la utilización de las variaciones de un objeto polimórfico se vuelven transparentes para el sistema, y como consecuencia, se disminuye notablemente el acoplamiento entre los componentes del proyecto.

La Indirección es recomendable para crear un objeto que pueda interactuar con otro objeto sin necesidad de un acoplamiento directo entre ambos. Entonces, si hay dos objetos que se comunican y uno o ambos cambian constantemente, se amerita utilizar la Indirección.

Al igual que con el Polimorfismo, al utilizar la Indirección, es sencillo reemplazar partes del sistema y probar con diferentes alternativas. En el caso específico de este simulador, esta ventaja podría ser incluso necesaria si se desea expandir el proyecto para probar y comparar diversas arquitecturas.

Un punto importante a considerar es que, al utilizar una Indirección, la lógica y dificultad de la implementación del proyecto aumenta. Al insertar una Indirección se agrega una capa de comunicación adicional entre dos componentes, lo que crea un intermediario que podría convertirse en un cuello de botella si la comunicación es muy frecuente. Por estas razones es necesario analizar cada situación y medir los costos antes de aplicar este patrón. Por ejemplo, como se mencionó anteriormente, se utilizó en el núcleo del procesador en este proyecto, y no en la administración de la memoria.

5.3. Factoría

Mediante el uso de este patrón, se puede apreciar una ventaja muy notable, la ocultación de la lógica y separación del sistema de la creación de Etapas y de los Registros MIPS. Al utilizar una Fábrica para crear las etapas que definen la arquitectura, es posible reemplazar por completo la arquitectura del procesador con solo modificar la implementación de la Fábrica. De esta forma, la lógica queda encapsulada en un solo punto, lo que facilita las pruebas e incrementa la mantenibilidad.

Otra ventaja es que respeta las responsabilidades de creación de los objetos, encargándose la Factoría de esto y evitando tener que incluir esta lógica de creación en varios objetos, lo que disminuye el acoplamiento.

La mayor desventaja de este patrón se da cuando varias partes del programa requieren utilizar los servicios de la Fábrica al mismo tiempo, en este caso se debe manejar la concurrencia apropiadamente para evitar inconsistencias.

5.4. Singleton

Al utilizar el patrón Singleton se logró dar acceso global a la Factoría FabricaDeEtapas mientras se mantiene un modelo orientado a objetos. La ventaja obtenida es que se simplifica el acceso a los servicios de la clase y se mantiene la consistencia. Al no tener que crear instancias de la FabricaDeEtapas se elimina la necesidad de lógica de creación de objetos y se obtiene un código más simple.

5.5. Observador

Este Patrón permite desacoplar la comunicación entre dos objetos y notificar sobre cambios en el sistema de una forma genérica. En el caso del simulador, se permitió crear una manera de avisar a la interfaz sobre los cambios en el estado de los componentes del simulador ocultando los detalles de implementación. De esta forma la interfaz queda desacoplada de la lógica del procesador y es sencillo reemplazarla por una interfaz distinta.

Además, como se mencionó anteriormente, los núcleos pueden trabajar a diferentes velocidades, por

lo que desplegar la información de cada núcleo apenas esta cambia es recomendable, y así se actualiza la interfaz sólo cuando es necesario.

Sin embargo, la mayor desventaja de este Patrón es el desempeño del sistema, el cual se reduce al utilizar Eventos para realizar la comunicación.

6. TRABAJO FUTURO

Para incrementar la capacidad de la aplicación a futuro, se puede aplicar nuevos Patrones en otras áreas del proyecto. Por ejemplo, el protocolo de control del bus usado en el proyecto actualmente es el protocolo "*Snooping*". Si se deseara implementar un nuevo protocolo para mejorar la eficiencia del bus con múltiples procesadores, se podría aplicar el patrón Estrategia y el patrón Composite para la implementación del protocolo de control de bus. De esta forma se podría experimentar con distintos protocolos sin necesidad de modificar drásticamente el código.

Por otro lado, en el área de administración de la memoria, se pueden utilizar patrones de manera similar a los utilizados en la creación de las Etapas y Registros MIPS. De esta forma, se puede implementar una FabricaDeMemoria que retorne instancias de distintas implementaciones de Caché y Memoria principal del proyecto. Cada una de estas clases serían polimórficas y poseerían implementaciones distintas que simulen distintos métodos para manejar el acceso a los datos persistentes. Las Etapas del procesador que necesiten acceder a la memoria necesitarían un Adaptador para recibir los distintos tipos de memoria que requieran.

7. CONCLUSIONES

Como se puede apreciar a lo largo de este documento, la utilización de patrones provee una flexibilidad que permite el desarrollo de aplicaciones más abiertas al cambio y a las actualizaciones. Sin embargo, estos beneficios podrían presentar algunas desventajas que deben considerarse. Los patrones de diseño pueden complicar el período de pruebas o la complejidad del código, y por ende queda a criterio de los desarrolladores analizar durante el proceso de modelado, el impacto de su aplicación.

Con la aplicación de patrones en el caso de estudio, se logró un diseño en donde todos sus componentes principales están desacoplados, lo que facilita el reemplazo de los mismos para extender la funcionalidad del simulador. Un beneficio adicional del desacoplo de estos componentes es una mayor facilidad para darle mantenimiento al proyecto, debido a que sus funcionalidades están encapsuladas y separadas. Por otro lado, se logró crear una interfaz no intrusiva a la lógica del simulador, permitiendo que el mismo se mantenga lo más fiel posible a la arquitectura en hardware que está simulando. Gracias a esto, la utilidad real del proyecto aumenta considerablemente, permitiendo con mayor facilidad aplicarlo a otras arquitecturas de computadores, o también crear una interfaz más amigable para el usuario.

A criterio de los autores, los patrones de diseño son excelentes prácticas que todo programador debería aplicar y que no pueden ser enseñados por métodos tradicionales, sino introducidos a los estudiantes de manera práctica.

BIBLIOGRAFÍA

Cinnéide M., Tynan R. (2004). A problem-based approach to teaching design patterns. En *ITiCSE-WGR '04 Working group reports from ITiCSE on Innovation and technology in computer science education*. (pp. 80-82). New York: ACM

Gestwicki P., Sun F. (2008). Teaching Design Patterns Through Computer Game Development. *Journal on Educational Resources in Computing (JERIC) JERIC Homepage archive*. *8*(1). Artículo 2. New York: ACM

Gómez, M., Jiménez, G., Arroyo, J. (2009). Teaching Design Patterns Using a Family of Games. En *ITiCSE '09 Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education*. (pp. 268-272). New York: ACM

Larman, C. (2004). *GRASP: Más patrones para asignar responsabilidades*. En L. Hernández (Trad.), UML y Patrones: Introducción al análisis y diseño orientado a objetos (pp. 305-320). New Jersey, USA: Prentice Hall.

Larman, C. (2004). *GRASP: Diseño de las realizaciones de casos de uso con los*

patrones de diseño GoF. En L. Hernández (Trad.), UML y Patrones: Introducción al análisis y diseño orientado a objetos (pp. 321-356). New Jersey, USA: Prentice Hall.

Von Neumann, J. (1945). *First Draft of a Report on the EDVAC*. En: Moore School of Electrical Engineering. University of Pennsylvania

Patterson, D., Hennessy, J. (2007). Computer Architecture, A Quantitative Approach. San Francisco, USA: Morgan Kaufmann Publishers.

Ritzhaupt, A., Zucker R. (2009) Evolutionary, not revolutionary, programming exercises using design patterns in an OO data structures course. *Proceedings of the 47th Annual Southeast Regional Conference*. Artículo 41. New York: ACM.

SOBRE LA Y LOS AUTORES

Isaac Montenegro Jiménez
Ingeniero en computación. Trabaja en Avantica Technologies en el departamento de Desarrollo móvil como ingeniero de sorftware. Correo electrónico:isaacmj@yahoo.com

Luis Rodríguez Rodríguez
Ingeniero bachiller en computación.
Correo electrónico: luisr90@gmail.com

Gabriela Salazar Bermúdez
Ingeniera Ciencias de la computación e informática. Trabaja como docente catedrática en la facultad de ingeniería de la Universidad de Costa Rica. Correo electrónico:gabriela.salazar@ecci.ucr.ac.cr