Modismos Basados en Tipos de Datos Atómicos de Java

JUAN CARLOS GONZÁLEZ TAMAYO, Posgrado en Ciencia e Ingeniería de la Computación, Universad Nacional Autónoma de México

JORGE LUIS ORTEGA ARJONA, Departamento de Matemáticas, Facultad de Ciencias, Universidad Nacional Autónoma de México

La programación concurrente y/o paralela juega un papel importante en el desarrollo de programas informáticos que realizan gran procesamiento de datos y/o que necesitan realizar acciones simultáneamente. Esta introduce una serie de conceptos, como los mecanismos de sincronización entre hilos y/o procesos, para lograr la exclusividad en el acceso a datos compartidos. Entre los más conocidos están los semáforos, regiones críticas, monitores, etc. Todos ellos, aunque muy útiles por lo general, presentan el problema de ser lentos en tiempo de ejecución a la hora de adquirirlos y liberarlos cuando hay gran contención. Son poco intuitivos de usar en la implementación de una aplicación paralela con múltiples componentes concurrentes que comparten datos. Y presentan varios problemas inherentes a su posible mal uso como los *deadlocks*, *livelocks*, etc.
A partir de que lenguajes de uso extendido, como Java o C++, han introducido bibliotecas que permiten leer y modificar algunos tipos de datos mediante operaciones basadas en instrucciones atómicas implementadas a nivel de *hardware* del tipo *read-write-update*, como *compare and swap*, se abre una nueva posibilidad de escribir algoritmos y estructuras de datos que pueden prescindir de los mecanismos de sincronización mencionados anteriormente.
El objetivo del presente trabajo es presentar dos patrones de bajo nivel o modismos basados en estas facilidades en el contexto de la programación concurrente y/o paralela. Primero se presenta el modismo *(*Atomic Flag), el mismo describe como resolver el problema de emplear variables booleanas para proteger secciones críticas. Finalmente se presenta el modismo *Atomic Counter*, el cual propone el empleo de enteros con operaciones atómicas para resolver el problema de un contador modificado de forma concurrente.

Términos Generales: modismos, operaciones atómicas

Palabas claves adicionales y frases: programación concurrente y/o paralela, Java *Atomic*

1. INTRODUCCIÓN

En el presente trabajo se presentan modismos basados en el uso de las variables atómicas de Java. El formato de presentación de patrones que se sigue es la forma POSA. Para los efectos del presente trabajo un modismo "es un patrón de bajo nivel que describe cómo implementar aspectos particulares de componentes de software o las relaciones entre estos con las herramientas de un lenguaje de programación dado" [Buschmann et al. 1996].
Primeramente, se describe el modismo *Atomic Flag*, el cual consiste en la definición y empleo conveniente de un objeto de tipo *AtomicBoolean* de Java como una bandera. La lectura y manipulación de la misma es *thread safe*.
Luego se describe el modismo *Atomic Counter*, como variable compartida que ejerce como contador. Esto se logra sin bloqueos a nivel de software, o sea, para realizar su cometido no es requerida la sincronización a nivel de programación. Por tanto, este contador no constituye una sección crítica en un programa concurrente.
Seguido se presenta el modismo *Atomic Counters Array*, que describe el empleo de un arreglo de enteros atómicos que puede ser usado, por ejemplo, en un algoritmo de ordenamiento por conteo (*Counting Sort*) concurrente.
Finalmente, se presenta el modismo *Atomic Node*, que presenta el concepto de nodo de una estructura de datos implementado con referencias atómicas y que permite su modificación y lectura de forma atómica.
Estos modismos hacen uso de variables con operaciones atómicas que provee el lenguaje de programación

Este trabajo es apoyado por el Consejo Nacional de Ciencia y Tecnología de México.
Contacto de los autores: Juan Carlos González Tamayo, email: jcgonzaleztamayo@gmail.com; Dr. Jorge Luis Ortega Arjona, email: jloa@ciencias.unam.mx

Java en su biblioteca estándar. Estas ideas se pueden portar al lenguaje de programación C++, que desde su especificación del 2011 incluye una biblioteca de atómicos similar a la que se emplea en el presente trabajo.

2. MODISMO *ATOMIC FLAG*

El modismo *Atomic Flag* es una forma libre de bloqueos, y a la vez, *thread safe* para leer y modificar una variable booleana desde dos o más componentes de *software* concurrentes y/o paralelos, que se ejecutan en una plataforma de memoria compartida. Es una técnica conocida que, en este caso, se presenta como modismo empleando objetos con operaciones atómicas de Java.

Ejemplo: Considere un componente *pipe* basado en el patrón *Shared Variable Pipe* (Figura 1) para la comunicación de procesos concurrentes o paralelos [Ortega-Arjona 2010]. Este componente *pipe* puede ser empleado como elemento de sincronización en el patrón arquitectónico de coordinación *Parallel Pipes and Filters*. Este patrón arquitectónico está basado en variables compartidas y comunicación asíncrona. En su presentación original, las *Shared Variable Pipe* se implementan usando monitores de Java para proteger un búfer compartido. El modismo *Atomic Flag* es empleando para implementar un *spinlock* en sustitución del monitor que protege el búfer. Por tanto, este spinlock es intercambiable con modismos de sincronización como los semáforos o regiones críticas y puede ser empleado para implementar monitores. ¿Cómo sincronizar el *Shared Variable Pipe* empleando *Atomic Flag*?

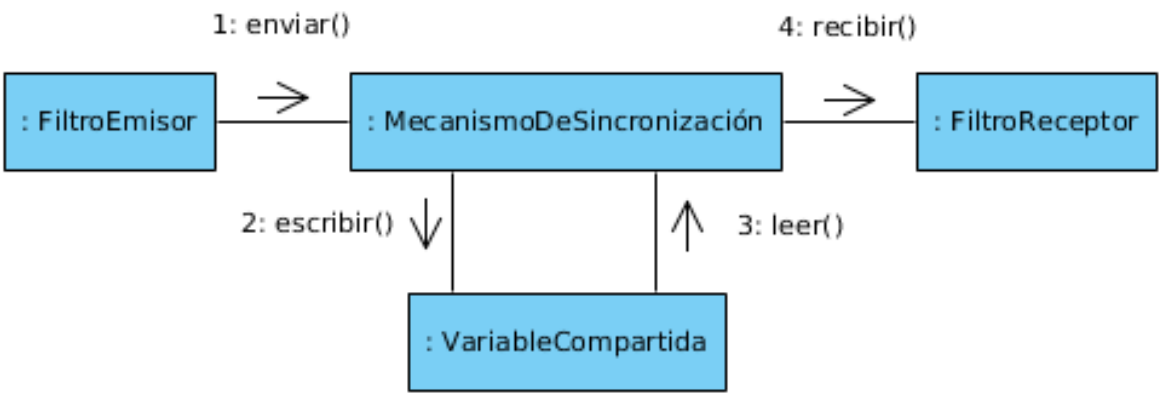

Fig. 1. Diagrama de colaboración del modismo *Shared Variable Pipe* [Ortega-Arjona 2010].

Contexto: Un programa concurrente y/o paralelo escrito en Java, que hace uso de variables booleanas compartidas para indicar estados del sistema o cualquier otro estado específico. Estas variables son leídas y modificadas por dos o más hilos de ejecución que compiten por acceder a ellas.

Problema: Para preservar la integridad de las variables booleanas compartidas, así como el orden de las operaciones sobre ellas, es necesario dotar a los componentes de *software* que se ejecutan en diferentes hilos de procesamiento de acceso seguro a dichas variables compartidas, para un número arbitrario de operaciones de lecturas y escrituras [Ortega-Arjona 2010].

Fuerzas: Para hacer un programa concurrente aplicando el presente modismo se deben tener en cuenta y balancear las siguientes fuerzas [Dijkstra 1968; Ortega-Arjona 2010]:

—El programa que se desarrolla requiere de estados binarios compartidos entre múltiples hilos de ejecución.

—Los hilos de ejecución del programa se ejecutan concurrentemente a diferentes velocidades relativas y de forma no determinista.

—Las operaciones de inspección y asignación para los propósitos de la sincronización son definidas como atómicas o indivisibles.

—La integridad de los valores de una variable compartida debe ser preservada siempre.

—El grado de concurrencia puede ser alto, o sea, el programa puede realizar un gran número de accesos al *Atomic Flag*.

Solución: Usar *Atomic Flag* para sincronizar de forma segura el acceso a las variables booleanas desde varios hilos de ejecución. Mediante su uso se espera como resultado hacer más veloz a aquellos programas paralelos escritos en lenguaje de programación Java que resuelven problemas enmarcados en el Contexto del modismo. Para crear un *Atomic Flag*, se crea un objeto de la clase *AtomicBoolean*. En este caso con nombre *flag*. El objeto (indistintamente llamado variable, ya que es lo que encapsula) *flag* es creado e inicializado con valor *true* de la siguiente forma:

```
AtomicBoolean flag = new AtomicBoolean(true);
```

Sobre la variable *flag* se pueden realizar las siguientes operaciones atómicas, tanto para su lectura como para para su modificación, definidas en la especificación del lenguaje [Corporation 2018]:

—Para modificar la variable sin importar el valor que tiene se emplea el siguiente enunciado:
```
flag.set(boolean newValue);
```
En este enunciado se llama al método *set* sobre *flag* y le pasa como parámetro el nuevo valor de *flag*.

—Para leer la variable se emplea el enunciado:
```
flag.get();
```
En este enunciado se llama al método *get* sobre *flag* el cual devuelve el valor actual de *flag*.

—Para modificar la variable sin importar el valor que tiene y obtener el valor que tenía antes de la modificación, se emplea el siguiente enunciado:
```
flag.getAndSet(boolean newValue);
```
En este enunciado se llama al método *getAndSet* sobre *flag* y le pasa como parámetro el nuevo valor de *flag*. El método *getAndSet* devuelve el valor anterior a la modificación.

—Para modificar la variable en dependencia del valor que tiene actualmente se emplea el siguiente enunciado:
```
flag.compareAndSet(boolean expectedValue, boolean newValue);
```
En este enunciado se llama al método *compareAndSet* sobre *flag* y se le pasan dos parámetros: el primero es el valor esperado que debe tener *flag* para poder ser modificado con el nuevo valor dado como segundo parámetro. El método *compareAndSet* devuelve un booleano indicando si se lleva a cabo la modificación o no.

Estructura: La figura 2 muestra el concepto de variable booleana atómica como un tipo de dato abstracto con un valor *booleano* y las operaciones descritas.

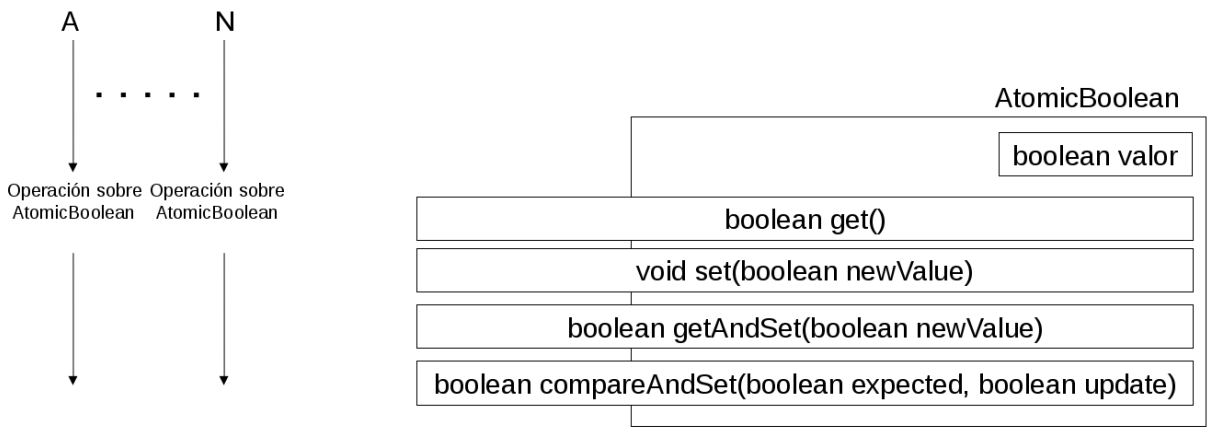

Fig. 2. Un diagrama representando un *AtomicBoolean* como un tipo de dato abstracto.

En el lenguaje de programación Java algunos usos típicos de este modismo se muestran en el siguiente fragmento de código:

```
import java.util.concurrent.atomic.AtomicBoolean;

public class JavaAtomicFlagExample {
    static AtomicBoolean flag = new AtomicBoolean(true);
    .
    public static void main(String args[]) {
    }
    public static void concurrentMethod1(){
        if (flag.get()) // Hacer algo
        else // Hacer otra cosa
    }
    public static void concurrentMethod2(){
        boolean condition = ...;
        flag.set(condition);
    }
    public static void concurrentMethod3(){
        boolean condition = ...;
        if (flag.getAndSet(condition)) // Hacer algo
        else // Hacer otra cosa
    }
    public static void concurrentMethod3(){
        boolean expected = ...;
        boolean condition = ...;
        while (!flag.compareAndSet(expected, condition)) { // Hacer algo }
    }
}
```

Dinámica: El *Atomic Flag* puede ser empleado principalmente en el siguiente escenario, caracterizado por la existencia de pocos procesos y regiones críticas de granularidad fina.

—Exclusión Mutua: En la figura 3 se presenta un diagrama de secuencia UML que muestra dos componentes, A y B, que comparten datos que requieren acceso exclusivo. El acceso a los datos compartidos está protegido por una variable atómica *mutex* inicializada con valor `true`.

El componente A adquiere el *mutex* asignándole valor `false`. Esto le da paso a la sección crítica. Luego el componente B intenta adquirir el *mutex* mediante una espera activa que termina cuando el componente A libera el acceso a la sección crítica asignando el valor *false*.

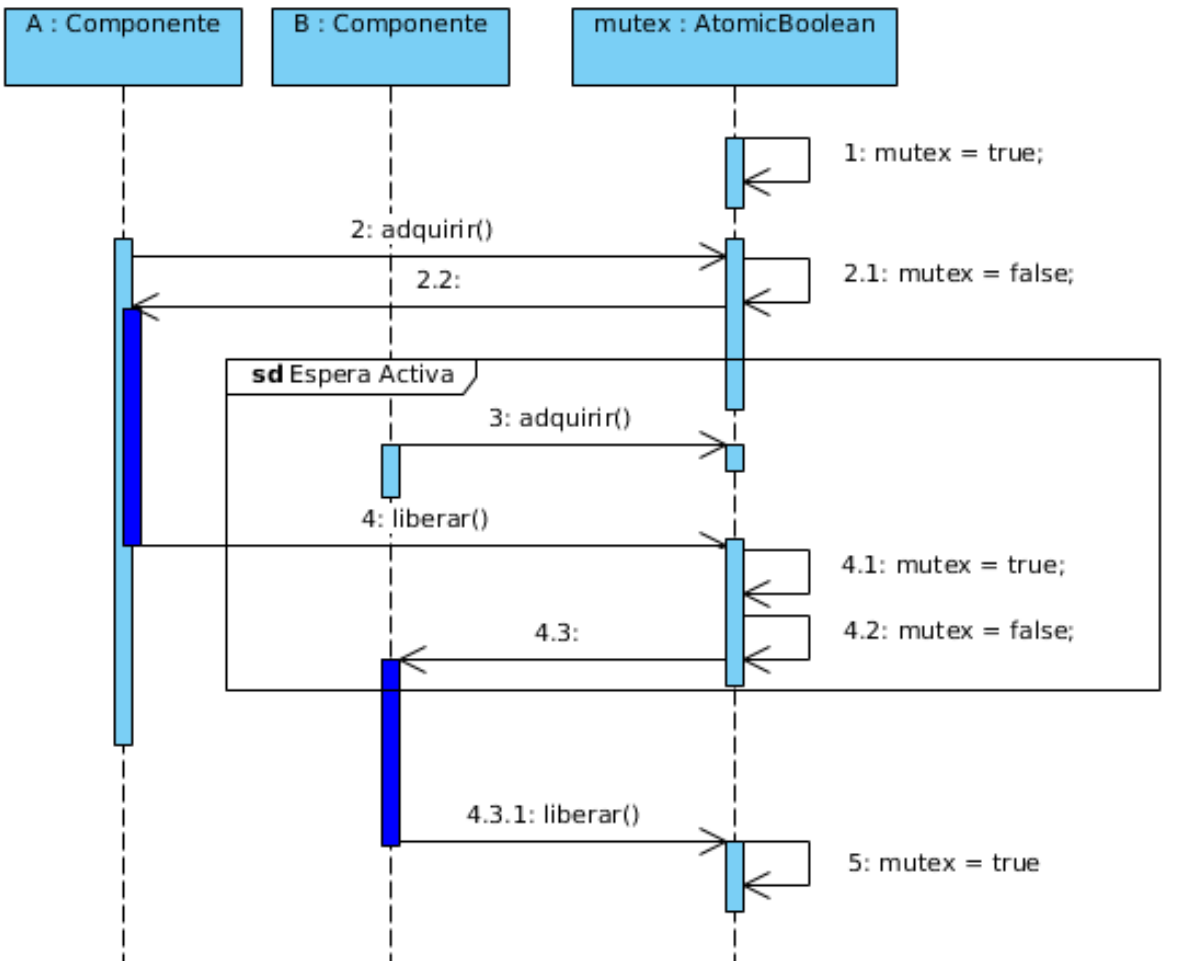

Fig. 3. Diagrama de secuencia de dos componentes de *software* concurrentes o paralelos que acceden a su sección crítica protegidos por un *Atomic Flag*.

Ejemplo Resuelto: El siguiente código muestra la implementación del patrón de diseño para el componente de comunicación *Shared Variable Pipe* empleando *Atomic Flag*, basado en un objeto de tipo *AtomicBoolean* y con nombre *mutex*. Como se aprecia, los métodos no están sincronizados al estilo Java, de tal forma que pueden haber más de dos hilos a la vez intentando progresar dentro de los métodos. En el caso de que el búfer esté lleno al intentar enviar o vacío al intentar recibir, se lanza una excepción, siempre liberando antes al *lock mutex*.

```
import java.util.ArrayList;
import java.util.concurrent.atomic.AtomicBoolean;

public class SharedVariablePipe {
```

```
    static final int MAXSIZE = 100;
    private ArrayList<Double> buffer = new ArrayList<Double>();
    AtomicBoolean mutex = new AtomicBoolean(true);
    public void send(double data) {
        while (!mutex.compareAndSet(true, false));
        if (buffer.size() == MAXSIZE) {
            mutex.set(true);
            throw new FullStackException();
        }
        buffer.add(data);
        mutex.set(true);
    }
    public double receive() {
        while (!mutex.compareAndSet(true, false));
        if (buffer.size() == 0) {
            mutex.set(true);
            throw new EmptyStackException();
        }

        double data = ((Double)buffer.remove(0)).doubleValue();
        mutex.set(true);
        return data;
    }
}
```

Usos conocidos: Algunos programas que hacen uso de *Atomic Flag* en [Herlihy and Shavit 2012] son:

—*Test And Set Spinlock*: *Spinlock* implementado como en el ejemplo presentado [Herlihy and Shavit 2012].

—*Test And Test And Set Spinlock*: *Spinlock* implementado con una operación *get* sobre el *Atomic Flag*. Este programa presenta granularidad fina y se corre en memoria compartida [Herlihy and Shavit 2012].

—*Semáforos binarios*: Los semáforos binarios se implementan con variables booleanas modificadas atómicamente. A diferencia de los *spinlocks*, se bloquea cuando la variable no tiene el valor esperado.

—*Variables que indican el estado de un recurso*: El típico uso de una bandera [Smirnov].

Consecuencias:
Ventajas:

—Mejora los tiempos de ejecución en casos de alta concurrencia.

—Es escalable en cuanto a que su funcionamiento es correcto, independientemente de la cantidad de hilos que lo emplean.

—No se utiliza ningún tipo de bloqueo a nivel de programación para leer y/o modificar la bandera. En consecuencia no se realiza ninguna llamada al sistema operativo.

Desventajas:

—En el caso de su uso como *spinlock*, su principal desventaja radica en el uso de la espera activa. Esta tiene un impacto mayor cuando el procesamiento en la sección crítica es de granularidad gruesa, debido a que la espera activa se vuelve más intensa. Es por ello que su uso debe ser evaluado. Para reducir el impacto de este problema, se debe evaluar el uso de la operación `yield()` en los hilos que esperan para que liberen el *CPU* para uso de los hilos que si están progresando. Se debe analizar la factibilidad de usar priorización dinámica de los hilos.

Patrones relacionados: El modismo *Atomic Flag* usado como *spinlock* está relacionado con los patrones de sincronización de hilos y/o procesos concurrentes y/o paralelos. Algunos patrones relacionados son el semáforo,

la región crítica y el monitor [Ortega-Arjona 2010]. También se relaciona con patrones también presentados en esta tesis como *Atomic Counter*.

3. MODISMO *ATOMIC COUNTER*

El modismo *Atomic Counter* es una forma de manejar contadores enteros sin sincronización a nivel de programación. Esto permite que dos o más componentes de software concurrentes y/o paralelos puedan llevar a cabo incrementos y/o decrementos simultáneamente sin bloquearse.

Ejemplo: Considérese un contador de instancias creadas de una clase desde el inicio de un programa paralelo en Java (figura 4), que puede ser manipulado de forma concurrente. No es un contador de instancias activas en memoria debido a que el método *finalize* de la super clase de Java *Object* no es seguro [Corporation 2018], por tanto, no se decrementa en el caso de que los objetos sean eliminados por el recolector de basura de la Máquina Virtual de Java. En el caso de C++, esto si es posible en tanto el programador tiene control total sobre los constructores y el destructor de la clase.

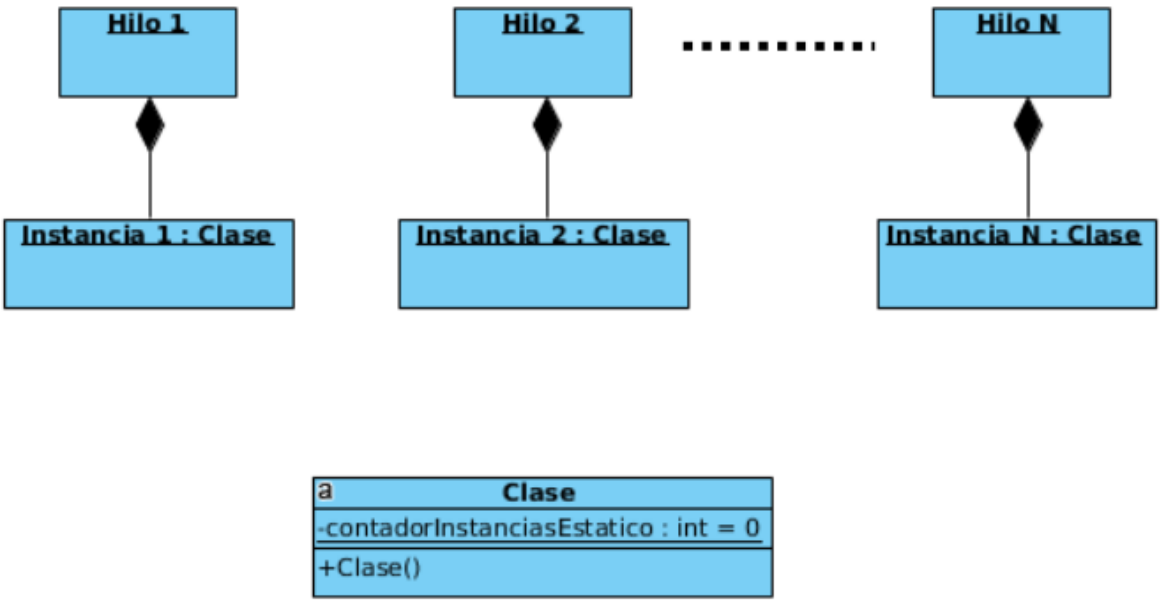

Fig. 4. Diagrama de objetos de un contador de instancias creadas. En el caso que se muestra, en cada llamada al constructor de Clase el atributo estático `contadorInstanciasEstatico` es incrementado. De tal forma, el valor del contador es N.

Contexto: Un programa escrito en algún lenguaje de alto nivel con soporte para variables con operaciones atómicas sobre ellas, como C++ o Java, en el cual se requiere el empleo de variables contadoras con actualización concurrente y libre de bloqueos.

Problema: Evitar condiciones de competencia que provoquen la pérdida de actualizaciones. O sea, preservar la integridad y consistencia de la variable contador.

Fuerzas: Para hacer un programa concurrente sincronizado mediante variables con operaciones atómicas se deben balancear las siguientes fuerzas:

—La variable contador puede ser leída y modificada por cualquier cantidad de hilos de ejecución.

—La variable contador solo puede ser incrementada, decrementada y leída.

—La solución debe ser intercambiable con contadores protegidos en las operaciones sobre él con modismos de sincronización como los semáforos, etc.

Solución: Usar variables con operaciones atómicas para lograr el efecto de un contador actualizado sin bloqueos a nivel de software. En el caso del lenguaje de programación Java tenemos las siguientes posibilidades [Corporation 2018]:

—Clase *AtomicInteger*: Clase que permite mantener un valor entero modificable con operaciones atómicas. O sea, no requieren sincronización a nivel de *software*.

—Clase *AtomicLong*: Clase que permite mantener un valor entero largo modificable con operaciones atómicas. O sea, no requieren sincronización a nivel de *software*.

En ambos casos contamos con las siguientes operaciones para modificar y leer la variable contador atómica [Corporation 2018].

—Método `int get()`: Devuelve el valor actual de la variable atómica.

—Método `int incrementAndGet()`: Incrementa de forma atómica en una unidad el valor de la variable y devuelve el nuevo valor incrementado.

—Método `int getAndIncrement()`: Incrementa de forma atómica en una unidad el valor de la variable y devuelve el valor anterior al incremento.

—Método `int decrementAndGet()`: Decrementa de forma atómica en una unidad el valor de la variable y devuelve el nuevo valor decrementado.

—Método `int getAndDecrement()`: Decrementa de forma atómica en una unidad el valor de la variable y devuelve el valor anterior al decrementado.

Las operaciones anteriores de incremento y decremento en una unidad son posibles de realizar de forma análoga con otros métodos que ofrece el *AtomicInteger*. Por ejemplo, la operación de incremento en uno puede lograrse además de las siguientes formas:

—Método `int accumulateAndGet(1, (x,y) -> x + y)`: Aplica de forma atómica la función lambda binaria `(x,y) -> x + y)` (en este caso suma los parámetros recibidos) tomando como parámetros el valor contenido en el objeto *Atomic Integer* y el primer parámetro del método `accumulateAndGet` (en este caso 1). Devuelve el valor luego de aplicar la operación.

—Método `int addAndGet(1)`: Suma de forma atómica el valor que se pasa por parámetro y devuelve el nuevo valor actualizado. Si se pasa uno se logra el efecto de incremento en una unidad.

—Método `int updateAndGet((x) -> x + 1)`: Aplica de forma atómica la función unaria (que recibe un único parámetro) que se pasa por parámetro al valor contenido en el entero atómico.

—Método `bool compareAndSet(int expectedValue, int newValue)`: De forma atómica compara el valor del entero atómico con el valor *expectedValue*, y de ser iguales lo actualiza con el valor del parámetro *newValue*. Para usar este método en un incremento se debe ser más cuidadoso que con las otras variantes ya que no es tan intuitivo. Sigue un ejemplo de cómo lograr un incremento empleando `compareAndSet`;

```
AtomicInteger ai = new AtomicInteger();
.
.
```

```
.
int currentValue;
int incrementedCurrentValue;
do
{
        currentValue = ai.get();
        incrementedCurrentValue = currentValue + 1;
}
while(!ai.compareAndSet(currentValue, incrementedCurrentValue));
```

Para el caso del decremento en una unidad, se puede derivar de las formas alternativas anteriores.

Estructura: La figura 5 muestra a una variable con operaciones atómicas en Java como un tipo de dato abstracto que tiene un valor y una interfaz con las operaciones de incrementar, decrementar y obtener valor.

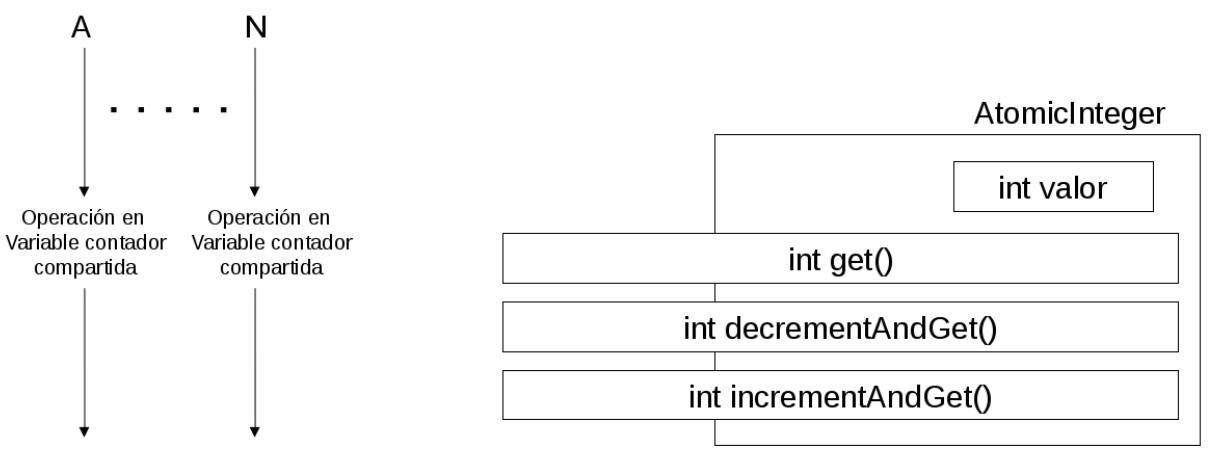

Fig. 5. Un diagrama representando un contador atómico como un tipo de dato abstracto.

A nivel de código el uso de las mismas es el siguiente:

```
import java.util.concurrent.atomic.AtomicInteger;

public class AtomicIntegerTester {
        static AtomicInteger ai = new AtomicInteger();
}

public void concurrentIncrement() {
        ai.incrementAndGet();
}

public void concurrentDecrement() {
        ai.decrementAndGet();
}

public int concurrentGet() {
        return ai.get();
}
```

Dinámica: El *Atomic Counter* puede ser empleado en cualquier escenario donde es necesario el empleo de un contador que puede ser actualizado de forma concurrente desde varios hilos. El diagrama de secuencia representado en la figura 6 muestra la interacción de dos componentes (A y B) con la variable entera atómica compartida av.

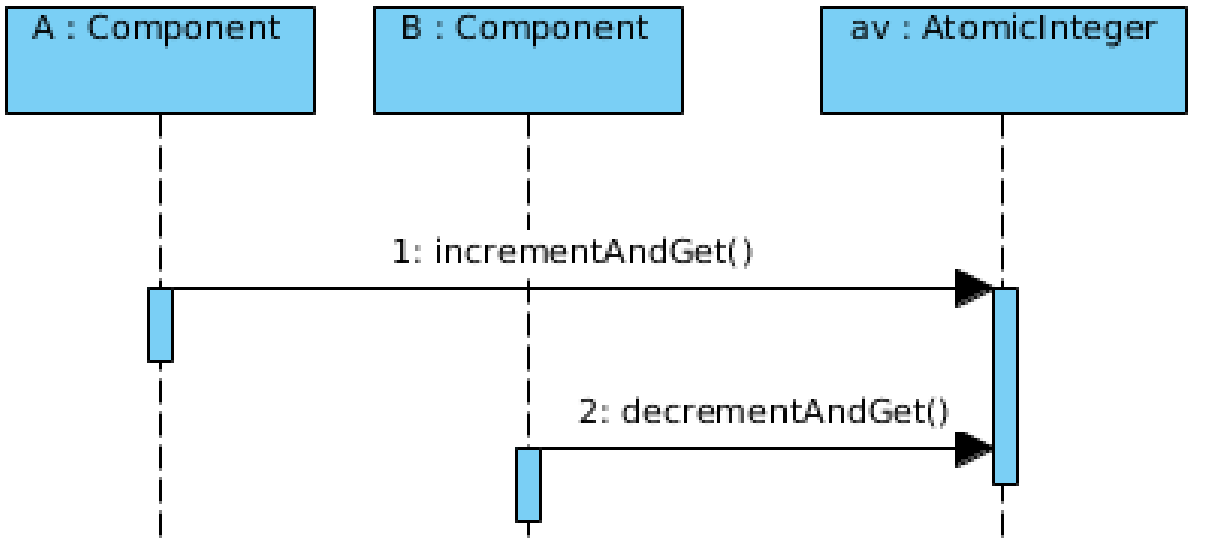

Fig. 6. Diagrama de secuencia la interacción de varios componentes de *software* con una variable entera atómica. Se aprecia que no hay sincronización a nivel de software en ningún momento. La cantidad de componentes puede ser n y las operaciones sobre la variable atómica pueden ser de cualquier tipo (incremento, decremento, obtención).

Ejemplo resuelto: A continuación, se muestra la implementación de una clase *MyObject* que hace uso de un contador atómico para contar las instancias creadas de la clase sin requerir sincronización desde la perspectiva del programador.

```
class MyObject
{
        static AtomicInteger instances = new AtomicInteger();
        MyObject()
        {
                instances.incrementAndGet();
        }
        public static int getInstances()
        {
                return instances.get();
        }
}
```

Usos conocidos: Algunos problemas que se pueden resolver empleando *Atomic Counter* son:

—Contador de referencias en un puntero de C++. O sea, que lleva la cuenta de cuántas referencias existen en el programa a la variable en cuestión. En tal caso de que este contador llegue a 0 se autodestruye la variable.

Este tipo de variables con contador de referencias se conocen también como punteros inteligentes. Esto implica un comportamiento parecido a un recolector de basura.

—Implementación concurrente de un patrón *Singleton*.

—Cualquier tipo de estadísticas de ejecución en un programa.

—Implementación de un semáforo general.

Consecuencias:

Ventajas:

—Mejora los tiempos de ejecución en casos de alta concurrencia con independencia de la cantidad de hilos de ejecución.

—Es escalable en cuanto a que su funcionamiento es correcto independientemente de la cantidad de hilos que lo emplean.

—Se elimina el concepto de sección crítica para la manipulación del contador atómico. Es decir no se requiere sincronización a nivel de *software* para actualizar el *Atomic Counter*.

—Posibilidad de sobrecargar los operadores preincremento ($++var$), postincremento ($var++$), predecremento ($--var$), postdecremento ($var--$) en el lenguaje C++.

Desventajas:

—Imposibilidad de sobrecargar los operadores preincremento ($++var$), postincremento ($var++$), predecremento ($--var$), postdecremento ($var--$) en Java, en tanto no lo permite el lenguaje.

Patrones relacionados: El modismo *Atomic Counter* está relacionado con el modismo *Atomic Flag*.

4. RESUMEN

En el presente capítulo se presentan cuatro modismos basados en tipos de datos con operaciones atómicas de la biblioteca *Atomic* de Java. Los modismos se presentan usando la forma POSA de [Buschmann et al. 1996].
En primer lugar se presenta el modismo *Atomic Flag*, el cual basado en un tipo de dato atómico permite la modificación concurrente y segura de una variable booleana. Esta puede usar como base en la implementación de un *spinlock*. También se emplea para representar estados del programa en el que se usen.
Luego se introduce el modismo *Atomic Counter*, que se basa en un valor entero modificado mediante operaciones de incremento o decremento atómicas. Este modismo puede ser usado en estadísticas de tiempo de ejecución de un programa, en la implementación de semáforos generales.

REFERENCIAS

BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P., AND STAL, M. 1996. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing.

CORPORATION, O. 2018. Documentacion Java JDK 9. https://docs.oracle.com/javase/9/index.html.

DIJKSTRA, E. W. 1968. Cooperating sequential processes. In *The origin of concurrent programming*. Springer, 65–138.

HERLIHY, M. AND SHAVIT, N. 2012. *The Art of Multiprocessor Programming, revised first edition*. Morgan Kaufmann.

ORTEGA-ARJONA, J. L. 2010. *Patterns for parallel software design*. John Wiley & Sons.

SMIRNOV, A. The missing article about qt multithreading in c.