ISSN 1870-4069

Propuesta de notación para el modelado de elementos de programación funcional en UML

Nanny Rodríguez-Munguía, Ulises Juárez-Martínez,
Gustavo Peláez-Camarena, Alma Sánchez-García

nrodriguezmunguia@acm.org, ujuarez@ito-depi.edu.mx,
sgpelaez@yahoo.com.mx, alivsaga@hotmail.com

Resumen. El paradigma de programación Objeto Funcional se ha extendido en diversos lenguajes de programación como son Scala, Java, C#, Ruby, entre otros; con ello surge la necesidad de que las herramientas de desarrollo permitan definir cómo abordar los nuevos conceptos en la codificación. Un ejemplo claro es UML, el cual tiene los medios necesarios para modelar los diversos contenidos del desarrollo de software, gracias a sus mecanismos de extensión; sin embargo, no posee un estándar para la modelación de los elementos de la programación Funcional, es por ello que en el presente trabajo se introduce una propuesta que permita definir el uso de elementos y propiedades funcionales, en el desarrollo de software con lenguajes Objeto Funcionales, como son: funciones de orden superior, funciones *currificadas*, evaluación perezosa, lambdas y mónadas.

Palabras clave: Programación funcional, programación orientada a objetos, lenguajes híbridos, paradigma orientado a objetos funcionales, Scala, desarrollo de software.

Propose of Notation for Functional Programming Modeling in UML

Abstract The Object-Functional programming paradigm it has been extended in various programming languages such as Scala, Java, C#, Ruby, among others. With this arise the necessity that the development tool allow to define how to approach the new concepts in the codification. A clear example is UML, which has the necessary means to model the diverse contents of software development, thanks to its extension mechanisms. However, it does not have a standard for the modeling of functional programming elements. But this is why presents in this work a proposal that allows define the use of properties and elements of the functional programming in Oriented Object-Functional languages, as are: high order functions, curried functions, laziness evaluation, lambdas and monads.

Keywords: Functional programming, oriented object programming, hybrid languages, paradigm oriented to functional objects, Scala, software development.

1. Introducción

En años recientes los lenguajes Orientado a Objetos, como Java y C#, han adoptado el paradigma de programación Funcional para brindar una mayor variedad de soluciones, que no eran posibles en este enfoque. Mientras que otros lenguajes, como Scala, han surgido considerando ambos paradigmas de programación desde su concepción.

En la actualidad el paradigma OO tiene amplio soporte para su modelado en UML [1], el cual no sólo proporciona los elementos necesarios para el modelado OO, sino que también posee mecanismos de extensibilidad que le permite modelar otros elementos que se integran dentro de un proyecto de software, como scripts, módulos, subsistemas y sistemas. Por otro lado el enfoque funcional no ha presentado madurez en el modelado de sus elementos; en este paradigma se identificó la metodología FAD [2], la cual proporciona elementos para el modelado de propiedades funcionales, pero no cumple con todas ellas, por lo que su soporte es limitado.

Aún con el amplio soporte de modelación OO que tiene UML, carece de estándares para el modelado de elementos funcionales que se encuentran inmersos en el enfoque de objetos. Es por esto que se propone una notación que sirva como estándar en el uso de propiedades funcionales en el lenguaje de modelado UML.

En el estado de la práctica se identificó que UML posee un amplio uso para la representación de los diversos enfoques que convergen en el desarrollo de proyectos de software: desde la modelación de elementos de hipermedia, hasta los nuevos paradigmas que mejoran la encapsulación de requerimientos no funcionales como lo es la programación Orientada a Aspectos. Esto demuestra que UML posee la capacidad ampliarse; sin embargo, sin un estándar, esta capacidad sólo le sirve a quienes adoptan su propia notación, como empresas, grupos e investigadores de área, por lo que limita la capacidad de compartir los modelos entre los diferentes grupos.

Al crear una notación estándar permitirá la generación de modelos que expresen las prácticas en el paradigma de la programación Objeto Funcional, con lo cual podrán abordarse independientemente del lenguaje de programación y así mejorar la difusión de soluciones.

2. Trabajos relacionados

Para el desarrollo de la propuesta, se realizó la investigación sobre los elementos y propiedades funcionales, así como propuestas de notación utilizando UML para diversos enfoques. A continuación se presentan los trabajos más relevantes.

En [2] se presentó la metodología de Análisis y Diseño Funcional (FAD) (del inglés Functional Analysis and Design), junto con una notación gráfica de elementos funcionales y sus relaciones. El soporte que brinda a nivel notación permite la modelación de tipos, herencia de tipos, funciones de orden superior, dependencia de funciones, funciones currificadas, módulos, subsistemas y sistemas; sin embargo, no aborda el uso de lambdas, mónadas ni evaluación perezosa. La metodología FAD abarca en su proceso la mejora de funciones, esto a través de técnicas que se basan en

la identificación de firmas de funciones durante el modelado, de manera que maximice la reutilización.

En [3] se aborda el uso de los mecanismos de UML para ampliar el lenguaje, de manera que permita el modelado de elementos ajenos al enfoque OO. En este trabajo se menciona que el mecanismo más utilizado es el estereotipo, por lo que se ha presentado un abuso en su utilización, y sugiere algunas medidas para mejorar su uso.
En [4] se describió la necesidad de las extensiones de UML, para el modelado de elementos de hipermedia utilizados en el desarrollo de software para la World Wide Web, de manera que brinde el soporte para crear estructuras de navegación. En este trabajo se presentó una propuesta para el diseño de hipermedia a través de uso de estereotipos, el uso del Object Constraint Language (OCL), mecanismos de extensión de UML.

En [5] se presenta una propuesta para el desarrollo de software utilizando el enfoque de la programación Orientado a Aspectos (AOP), utilizando UML y sus mecanismos de extensión para representar los asuntos de corte. Esta propuesta también se apoyó en el uso de estereotipos para describir los diferentes tipos de aspectos y el modelado del entrelazado de código. Este trabajo muestra que UML es capaz de extenderse para representar enfoques de programación que se integran en el paradigma OO.

El estado de la práctica indica que FAD no tiene el soporte suficiente para integrarse con el Modelado de Objetos, mientras que UML posee múltiples muestras de su capacidad para abordar los diferentes enfoques en el desarrollo de software. Por lo tanto, es factible realizar una propuesta que permita integrar los elementos de la programación Funcional, utilizando los diferentes mecanismos de UML y su estructura.

3. Notación propuesta

El Enfoque Funcional posee diversas propiedades, algunas no son exclusivas del paradigma (como el polimorfismo paramétrico), mientras que otras se encuentran inmersas en el lenguaje de programación (como es la tipificación fuerte y ciudadanos de primera clase); la propuesta que aquí se presenta, permitirá representar los elementos funcionales como son funciones de orden superior, funciones currificadas, evaluación perezosa, mónadas y el uso de lambdas; de manera que se introduzcan en las fases de modelado de software y finalmente su codificación. Los ejemplos que se muestran a continuación presentan su codificación en lenguaje de programación Scala.

3.1. Funciones de orden superior

Las funciones de orden superior son aquellas que tienen la capacidad de recibir una o más funciones como argumentos, o bien, devuelven una función como valor de retorno; esto recae en la necesidad de representar este tipo de argumento poco usual: una operación (método o función). Para el Enfoque Funcional, todas las funciones poseen una firma [2], comúnmente representada a través del símbolo -> que separa cada argumento y el último indica el valor de retorno. Para el enfoque OO, en la notación UML [1], una operación se define a través de un nombre seguido de un paréntesis

dentro de los cuales se presentan los argumentos que recibirá con un nombre y su tipo separado por dos puntos, cada argumento separado por comas, junto a los paréntesis se usan dos puntos y el tipo de retorno.

Dado que en la notación de UML no se requiere especificar un nombre para los argumentos, la notación propuesta es: f(type, type, ..)->type , donde f es un alias para la función, entre paréntesis los tipos de datos que recibe, separados por comas seguido del símbolo ->, el cual señala al tipo de retorno. En la figura 1 se muestra un ejemplo, donde operation1 recibe una función f que tiene como parámetros un valor de tipo int y devuelve otro valor de tipo int; mientras que operation2 recibe un parámetro de tipo boolean y devolverá una función f cuyos parámetros son: el primero de tipo String y el segundo de tipo int, devolviendo un tipo double.

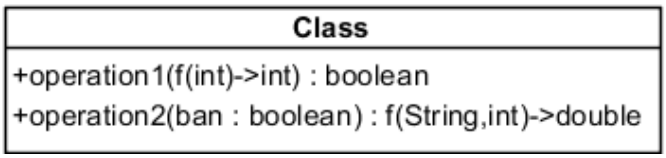

Fig. 1. Ejemplo de notación para funciones de orden superior.

La implementación en código del ejemplo de funciones de orden superior es el que se muestra a continuación en el algoritmo 1, obsérvese que dentro del método operation2 se utilizan funciones al vuelo (lambdas), las cuales se revisan más adelante.

Algoritmo 1. Implementación en código de la clase Class y los métodos operation1 y operation2.

```
class Class{
  def opertation1(f:(Int)=>Int):Boolean={
    return f(8)>100;
  }

  def operation2(ban:Boolean):(String, Int)=>Double={
    if(ban)
      return ((a:String, b:Int)=>a.length()/b);
    else
      return ((a:String, b:Int)=>a.length()*b);
  }
}
```

Esta notación permite claridad en la separación de los tipos que son argumentos, del tipo de valor de retorno, aunque esta notación casi coincide con la sintaxis usada en el lenguaje Scala, para definir una función como argumento de otra función, se diferencia únicamente por el símbolo -> por =>. En el caso de una función como valor de retorno, es posible omitir el alias, puesto que puede usarse de manera anónima o designarle un nuevo nombre dependiendo de su implementación.

3.2. Funciones currificadas

Una función se denomina currificada, cuando recibe sus parámetros uno a la vez; en los lenguajes OO esto no ocurre de manera común, solo algunos lenguajes soportan esta propiedad. En Scala [7] es posible definir una función currificada de forma variable, es decir, cuántos y qué parámetros recibirá en cada llamada parcial, por ello es necesario especificar la distribución de los argumentos y el orden de las posibles llamadas parciales; la notación propuesta para esta propiedad es definir la separación de argumentos con paréntesis, de manera que se separen estas llamadas parciales. Como se muestra en la figura 2, se tiene al método operation1(a:int , b:int, c:int)(d:int, e:int)(f:int):boolean, el cual en su primera llamada parcial debe contener tres argumentos de tipo int, esto retornará una operación anónima que recibe tres parámetros de tipo int, que de igual forma es posible llamar parcialmente, enviando dos parámetros de tipo int, retornando otra función anónima que tendrá como parámetro un tipo int, al llamar esta última función, con el parámetro faltante, se completará la llamada y retornará el resultado de tipo double. En los otros ejemplos operation2 y operation3, se muestra que la distribución de currificacion se puede especificar de forma variable, agrupando argumentos de uno en uno o más. Nótese que operation3 recibe como argumento currificado una función f(int)->boolean; de hecho las funciones currificadas también son funciones de orden superior, con el simple hecho de retornar funciones, y ahora también al recibirlas como argumento. En el algoritmo 2 se muestra el código correspondiente a este ejemplo.

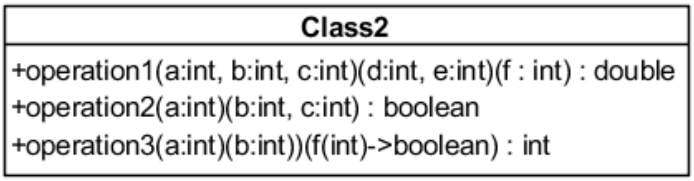

Fig. 2. Ejemplo de notación para funciones currificadas.

Algoritmo 2. Implementación en código del modelo de la clase Class2 y sus respectivos métodos.

```
class Class2 {
  def operation1(a:Int, b:Int, c:Int)(d:Int, e:Int)(f:Int):  Double = {
        return a + b + c / (d * e) - f;
    }
  def operation2(a:Int)(b:Int, c:Int): Boolean = {
        return a > (b - c)
    }
  def operation3(a:Int)(b:Int)(f:(Int) => Boolean): Int ={
        if (f(a))
            return a + b;
   else
```

```
            return a - b;
  }
}
```

A pesar de que las llamadas parciales retornan funciones, no es necesario especificar esto, puesto que se sobreentiende que al aplicar la currificación con llamadas parciales, la función devuelta será la misma pero con los argumentos faltantes para completar la llamada a dicha operación.

3.3. Evaluación perezosa

La evaluación perezosa permite que los valores se evalúen únicamente cuando son necesarios, esta estrategia permite la manipulación de estructuras de datos infinitas, denominadas flujos (*streams*). Dado que la evaluación perezosa se aplica sobre valores, la notación que se propone debe aplicar sobre los atributos de una clase.

UML tiene soporte para el nivel de visibilidad de dichos atributos, entre los cuales se hace uso de los símbolos # (protegido), - (privado), + (público) y ~ (paquete), por lo que estos símbolos se descartan como opciones de notación. Sin embargo, el símbolo "~" no se usa frecuentemente, y es uno de los símbolos ASCII que se asocia fácilmente a la palabra flujo, por ello la notación propuesta es hacer uso de los símbolos :~ de forma conjunta donde ":" significa evaluación y "~" significa flujo, dando a entender que dicho atributo se evaluará de forma que mejore su desempeño, esto da apertura al uso de la evaluación perezosa.

En la figura 3 se muestra la clase InfineCalculus, la cual contiene dos métodos de cálculo infinito fibonacci() y count(), la clase LazzinesEvaluation, posee 3 atributos que serán evaluados de forma perezosa, fibonacciResults (es público), numbers (es visibilidad de paquete), y x (privado); los dos primeros indican que serán variables auxiliares en la obtención del cálculo fibonacci() y de count().

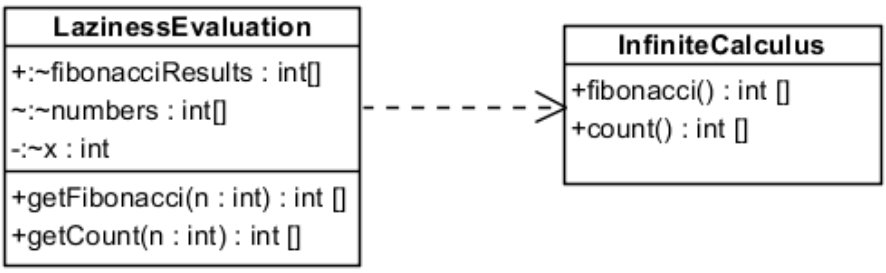

Fig. 3. Ejemplo de notación para la evaluación perezosa.

Nótese que el atributo x, también se evalúa de manera perezosa, sin embargo, no es un arreglo de int, esto es debido a que no contendrá resultados de los métodos de InfiniteCalculus, mostrándose que cualquier atributo tiene la capacidad de evaluarse de forma perezosa, y no es exclusivo de una estructura de datos. En el algoritmo 3 se muestra la implementación en código de Scala, del modelo de ejemplo para la

evaluación perezosa. En este ejemplo se demuestra que la notación propuesta no se confunde con la de visibilidad de paquete, estándar de UML.

Algoritmo 3. Implementación en código del modelo presentado en la fig. 3 para la evaluación perezosa de cálculos infinitos.

```
class InfiniteCalculus {
  def count(): Stream[Int] = {
    Stream.cons(0, count() map (_ + 1));
    }
  def fibonacci(): Stream[Int] = {
    Stream.cons(0, (0 #:: fibonacci.scanLeft(1)(_ + _)));
    }
}
class LazinessEvaluation{
  val cal=new InfiniteCalculus();
  lazy val fibonacciResults=cal.fibonacci();
  lazy protected val numbers=cal.count();
  lazy private val x={println("Hola");100}

  def getFibonacci(n:Int):Array[Int]={
    fibonacciResults.take(n).toArray
  }
  def getCount(n:Int):Array[Int]={
    numbers.take(n).toArray
  }
}
```

3.4. Mónadas

De acuerdo con [8], las mónadas son un mecanismo de la programación funcional que permite la introducción de declaraciones imperativas y proporciona una manera de abstraer sobre diferentes tipos de cálculos. Un cálculo se considera como una función que típicamente producirá un valor, cada cálculo se caracteriza por una estructura específica de los parámetros y valores de retorno, al definir un constructor de tipos, se define el tipo de este cálculo. Un ejemplo es la mónada *Maybe*, que define cálculos que se caracterizan por producir un valor, pero es posible que falle, los tipos de esta clase de cálculo se define como *Just* y *Nothing*, donde *Just* es el valor producido, y *Nothing* es el caso de fallo.

Dada esta definición, los cálculos de este tipo pueden ser unidos secuencialmente, el valor se inyecta extrayéndolo de su contexto actual y enlazándolo a otra función que acepta dicho valor. Para realizar el enlace es necearia una función que permita mapear el valor actual de la mónada hacia otra función del mismo tipo, que en los lenguajes funcionales es denominado *bind***.** Dada esta definición, un tipo monádico, en lenguajes Objeto Funcionales, es una clase que posee la encapsulación de un valor en cierto

contexto, y tiene el comportamiento necesario para mapear o enlazar dicho valor fuera de ese contexto, de manera que se crea un nuevo objeto perteneciente al mismo tipo.

La representación que se muestra en la figura 4 es la mónada *Maybe*. Este modelado se apoya en la notación propuesta para funciones de orden superior para definir el método que funcionará como el enlace, en este caso map(); la notación estándar de UML para representar jerarquía, que representa las dos posibles construcciones de Maybe; y clases genéricas para la referencia de un valor en dado contexto. El código resultado de este modelo se muestra en el algoritmo 4, cabe destacar que para la definición de una estructura monádica, los lenguajes se apoyan de diferentes características del propio lenguaje, como en Scala es el uso de *sealed trait*, *object*, y *case*.

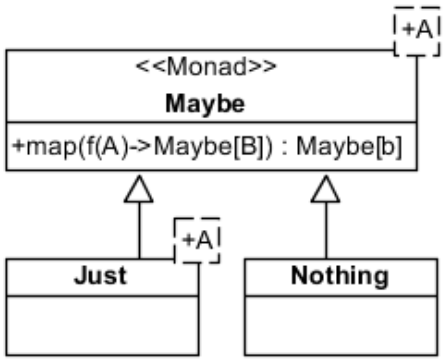

Fig. 4. Ejemplo de notación para una mónada.

Algoritmo 4. Código fuente correspondiente a la mónada Maybe.

```
sealed trait Maybe[+A] {
  def map[B](f: A => Maybe[B]): Maybe[B] = this match {
    case Just(a) => f(a)
    case Nothing => Nothing
  }
}
case class Just[+A](a: A) extends Maybe[A] {
  override def map[B](f: A => Maybe[B]) = f(a)
}
case object Nothing extends Maybe[Nothing] {
  override def map[B](f: Nothing => Maybe[B]) = Nothing
}
```

La definición que se muestra en la figura 4 requiere un mayor nivel de detalle; por otro lado, no se asevera que todas las mónadas se apoyen del uso de genéricos, además de que la especificación del mapeo variará en función a las necesidades. Por ello se propone el uso del estereotipo <<Monad>>, en adición a la representación de la mónada, de esta manera, se hace referencia a este tipo de construcción de forma más directa; el uso de genéricos y la definición de métodos para el mapeo, será parte del

refinamiento de dicho modelo. En la figura 4 se muestra el uso del estereotipo propuesto.

3.5. Uso de lambdas

Una de las propiedades funcionales cuya notación, a nivel de especificación de UML, varía dependiendo de la forma en que se implementa, son las expresiones lambda. Las expresiones lambda son funciones anónimas que se implementan cuando son necesarias, estas se utilizan comúnmente cuando se envían como parámetro a otras funciones. Definir el uso de funciones anónimas en un modelo UML implica describir el comportamiento interno de un método, sin embargo, ese nivel de descripción no es común.

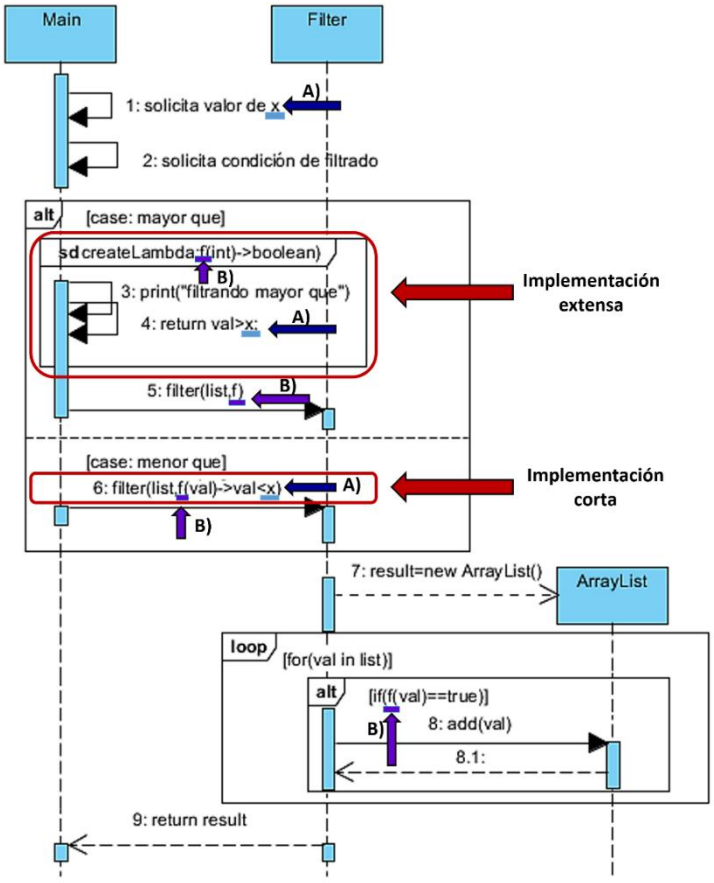

Fig. 5. Ejemplo de modelado para el uso de lambdas.

Una forma factible para describir la creación de una función anónima es a través de un diagrama de interacción, donde se observa la llamada y los argumentos que se envían

a un método, por lo que se hace referencia al comportamiento dinámico del sistema. Esta forma de modelación es especialmente útil si existen diferentes versiones de dicha función en casos alternos.

En la figura 5 se muestra un caso particular, donde la clase Filter posee una función de orden superior, llamada filter(), la cual recibe dos parámetros: una lista y una función; y realiza un filtrado de dicha lista a partir de la función recibida. La clase Main solicita un valor para x, y el tipo de filtrado a realizar, de manera que, de acuerdo a la selección, se creará una función anónima para cumplir dicho requisito.

En el caso de implementación extensa, se hace uso de un elemento *frame* de UML, en el cual se especifica createLambda:f(int)->boolean, donde se hace referencia a la creación de una lambda cuya firma de método es la señalada después de dos puntos.

Este *frame* es necesario, debido a que una implementación extensa requiere más de una instrucción; en este caso, se imprime "filtrando mayor que", cada vez que se hace la llamada al método, y la última instrucción señala el valor de retorno, que es la evaluación de mayor qué entre val (un alias del valor a recibir) y el valor de x (un valor ya definido en el objeto actual, inciso A). Finalmente se hace la llamada a la función de orden superior filter(), al cual se le envía como parámetro el alias de la función definida en el *frame* (ver inciso B).

En el caso de la implementación corta, simplemente se hace la llamada de la función filter(), enviando la lista y de forma breve la función anónima, de forma similar a la firma del método, señalando el alias de la función, seguido de paréntesis (dentro de los cuales se definen los parámetros a recibir) , el símbolo -> que señala a la instrucción a realizar, en este caso la evaluación menor qué entre val y x.

Es importante mantener la consistencia de los alias de función en el diagrama, puesto que de no hacerlo dificultará su interpretación. Por otro lado, en la implementación no es forzoso mantener dicha consistencia, sin embargo, es aconsejable para facilitar la lectura del código.

En el algoritmo 5 se muestra cómo es la implementación del ejemplo mostrado en el diagrama de la fig. 5, lo que demuestra que el modelo es factible para la programación en el lenguaje Scala, dado que el uso de lambdas es trasparente para el programador (todos los lambdas implementan de la clase Lambda); por lo que la base para su implementación es definir su firma y su proceso interno. Sin embargo en el lenguaje Java (versión 8), el uso de lambdas es a través de interfaces funcionales; por lo que es necesario señalar la interfaz funcional que implementa el lambda que se define; esta especificación es parte del refinamiento del modelo, por lo que se parte inicialmente con la definición del lambda en el diagrama de secuencia, y posteriormente crear las interfaces funcionales necesarias, para cumplir con las firmas de función.

Algoritmo 5. Código correspondiente al diagrama de secuencia de la fig. 5.

```
class Filter {
  def filter(list: List[Int], f: (Int) => Boolean): ArrayList [Int] = {
    var newList = new ArrayList[Int]();
    for (valor <- list) {
      if (f(valor) == true)newList.add(valor)
```

```
    }
    return newList;
  }
}
object Main {
  def main(args: Array[String]) {
    val x = readLine().toInt
    var list = List(1, 2, 3, 4, 5, 6, 8);
    print("introduce condición de filtrado");
    readLine() match {
      case "mayor que" => { //implementación larga
        def f = (valor: Int) => {
          print("filtrando mayor qué")
          (valor > x)
        }
        new Filter().filter(list, f);
      }
      case "menor que" => new Filter().filter(list, (valor => valor < x)); //implementación corta
    }
  }
}
```

4. Conclusiones y trabajo futuro

La notación propuesta para representar elementos funcionales es completamente compatible con UML, dado que se apoya de los elementos que proporciona este lenguaje de modelado, y también se asocia a lo nomenclatura utilizada en el enfoque funcional. Los ejemplos utilizados demuestran que es viable utilizar esta notación en aplicaciones reales, puesto que todos los ejemplos fueron realizados en *Visual Paradigm*, una herramienta para el modelado con UML.

Como trabajo a futuro se espera evaluar la notación a través del desarrollo de proyectos en los distintos Lenguajes Objeto Funcionales, de manera que no presente inconsistencias al realizar la codificación.

Referencias

1. Rumbaugh, J., Jacobson, I., Booch, G.: El Lenguaje Unificado De Modelado, Manual De Referencia. Segunda edición Madrid, Pearson Educación, S.A (2007)
2. Russell, D. J.: A Functional Analysis and Design Methodology. Ph.D. dissertation, Universidad de Kent en Canterbury (2000)
3. Baumeister, H., Koch, N., Mandel, L.: Towards a UML extension for hypermedia design. In: International Conference on the Unified Modeling Language, Springer Berlin Heidelberg, pp. 614–629 (1999)

4. Henderson-Sellers, B., Gonzalez-Perez, C.: Uses and abuses of the stereotype mechanism in UML 1.x and 2.0. In: International Conference on Model Driven Engineering Languages and Systems, Springer Berlin Heidelberg, pp. 16–26 (2006)
5. Chiusano, P., Bjarnason, R.: Functional Programming in Scala. 1st ed, Manning Pubications Co. (2012)
6. Ballal, R., Hoffman, M. A.: Extending UML for aspect oriented software modeling. In: Computer Science and Information Engineering, 2009 WRI World Congress on, IEEE, Vol. 7, pp. 488–492 (2009)
7. Hoefer, C., Ostermann, K.: On the relation of aspects and monads. In: Sixth International Workshop on Foundations of Aspect-Oriented Languages, FOAL (2007)