Director
Teoría de la Computación

Ulises C. Ramirez [ulir19@gmail.com, ulisescolrez@gmail.com]
Héctor J. Chripczuk [hectorejch@gmail.com]

01 de Noviembre, 2018

Versionado

Para el corriente documento se está llevando un versionado a fin de mantener un respaldo del trabajo y además proveer a la cátedra o a cualquier interesado la posibilidad de leer el material en la última versión disponible.

Repositorio: *https://github.com/ulisescolina/UC-TC*

–Equipo Director

Índice general

Índice de Fragmentos de Código

Capítulo 1

Estado de la Cuestión

1.1. Introducción

En el campo del desarrollo de software el advenimiento de notaciones como UML generaron un nuevo paradigma para la implementación de software, si bien esta notación no es una idea nueva, por el hecho de que tiene sus inicios en la decada de los 1990 siendo la idea principal del mismo, el modelado de sistemas de software en varios niveles de abstracción [1], además considerandose por estudios como la herramienta de notación para modelado de software dominante [2] aunque usado predominantemente de manera informal y que las actividades realcionadas al ámbito tienden a ser más fáciles en un desarrollo con un enfoque en el modelado [3] con la capacidad de además de tener beneficios dentro del entorno de desarrollo dado al nivel de expresión que se tiene en las notaciones gráficas comparadas con las formas textuales de expresión [4].

A pesar de lo mencionado, encuestas realizadas en uno de los estudios que trata a UML como una notación dominante dentro del desarrollo de software realizada años atras, muestran que desarrolladores prefieren centrar el trabajo en vastas cantidad de líneas de código, raramente mirando diagramas, y mucho menos confeccionándolos o editándolos, posibles razones para esto se pueden para este comportamiento se descubre en [5] en donde se identifican cuatro razones por las cuales los efectos positivos de UML pueden verse reducidos, aunque para el presente trabajo se puede hacer hincapié en tres de los mencionados: *la no-factibilidad de realizar ingenieria inversa en codigo heredado*, *costo de entrenamiento del equipo* y *los requerimientos no estaban distribuidos en unidades funcionales del sistema*.

En el mismo estudio, se consultó cuáles son las cuestiones de trabajar con el desarrollo orientado a modelos que lo hacen difícil de tratar o sean la razon por la cual no se modele mucho. Las razones principales fueron:

1. *Los modelos se desactualizan con respecto al código*, es decir no se encuentra un flujo de trabajo que contemple a ambas actividades tal que estas dos avancen a la par, sino que se deben realizar por separado, causando

así que el modelo quede desactualizado con frecuencia.

2. *Los modelos no son intercambiables fácilmente*, aquí se habla de la dificultad de, nuevamente, encontrar un flujo de trabajo que permita el paso de los modelos de forma más "natural".

3. *Las herramientas para modelado son pesadas*, haciendo alusión a los recursos computacionales en los cuales se debe incurrir para utilizar dichas herramientas.

4. *El código generado desde una herramienta no es de mi agrado.*

5. *Crear y modificar un modelo es lento.*

Además de la encuesta anterior, también se realizó una centrándose en el desarrollo *centrado en la codificación*, los siguientes fueron los mayores inconvenientes identificados por los desarrolladores.

1. Ver de "un pantallazo" todo el diseño es un problema, una forma de transmitir la dificultad que se tiene de brindar una visión a un alto nivel del proyecto, lo cual lleva al tercer punto de la lista.

2. Comportamiento del sistema difícil de entender.

3. La calidad del código se degrada con el tiempo.

4. Muy dificil reestructurar el proyecto.

5. Cabios de codigo pueden implementar bugs.

1.2. Desarrollo de Software Dirigido por Modelos

El desarrollo de software dirigido por modelos (MDSD, de ahora en mas, por sus siglas en inglés) como se menciona en [6] , surge a partir de la popularización de UML, el uso del mismo, sin embargo, solamente se restringía a la confección de documentación, debido a motivos ya mencionados, el acercamiento de que ofrece MSDS es enteramente diferente, la parte "dirigido" (Driven, en MDSD), enfatiza la importancia y el rol central que le da este paradigma al modelo éste ya no solamente constituye la documentación del software, sino que es considerado igual a código, este es comparable a campos de alta especialidad, dada una de las características del acercamiento, esta es que apunta a la realización de abstracciones específicas de dominio y a hacer estas abstracciones accesibles a través del modelado, ésta característica permite la automatización de la implementación de código, haciendo posible a la vez el incremento en la productividad y la mantenibilidad de los sistemas.

Para que se pueda aplicar el concepto de "modelo específico de dominio" se deben tener en cuenta tres requerimientos:

- *Lenguajes de dominio específico*, para la formulación de modelos.
- Lenguajes que puedan expresar transformaciones "modelo-código".
- *Compiladores, generadores o transformadores*, para generar el codigo ejecutable en varias plataformas.

1.3. Arquitectura Dirigida por Modelos

La arquitectura dirigida por modelos (MDA, por sus siglas en ingles) es una iniciativa introducida por el Object Management Group, con el proposito de brindar una forma estandarizada para la especificación e interoperabilidad de sistemas basado en el uso formal de modelos [7], en el núcleo de MDA se encuentran otros estándares implementados por el OMG: the Unified Modeling Language (UML), Meta Object Facility (MOF), XML Metadata Interchange (XMI) y el Common Warehouse Metamodel (CWM). Al igual que el MDSD, MDA, ubica los modelos de sistemas en el nucleo del problema de interoperabilidad lo cual hace que la implementación del sistema sea independiente de la técnología.

La OMG, promueve MDA como un *marco de trabajo arquitectónico para el desarrollo de software*, el cual esta construido alrededor de un numero de especificaciones detalladas de la misma organización que son usadas ampliamente en la comunidad de desarrolladores.

Esto puede hacer pensar que $MDA = MDSD$, lo cual sería correcto hasta cierto punto, en principio, el acercamiento de MDA es similar al de MDSD, pero difiere en detalles, por ejemplo, éste, tiende a ser mas restrictivo, enfocándose primariamente en lenguajes de modelado basados en UML [6].

1.4. Lenguaje Específico de Dominio

Un lenguaje especifico de dominio (DSL) es un lenguaje de software el cual se especializa en abarcar un problema dentro de la ingeniería en particular, las cuales son características para un dominio de aplicación, éste, se basa en abstracciones alineadas a este dominio y provee una sintaxis propicia para aplicar estas abstracciones de forma efectiva [8].

La implementación de un DSL permite mitigar ciertos aspectos que se vieron como desventaja al inicio, algunos mencionados en [9] incluyen la reutilización de código, promueve la legibilidad y el entendimiento debido al alto nivel de abstracción del mismo, permite a que usuarios con nivel bajo en programación la creación de modelos para programas siempre y cuando estos posean el conocimiento del dominio, más verificaciones en la sintáxis y semantica que un lenguaje de modelado general; aunque también se enfatizan desventajas que estos insertan en el desarrollo, curva de aprendizaje necesaria y la falta de personas letradas en el DSL, ya que es mas probable que las personas sepan como

resolver los problemas adoptando un lenguaje de propósito general el cual ya conocen.

Algunos autores definen las características deseables de un DSL [10]:

- La capacidad de describir todo el software.
- La capacidad de describir varios niveles de abstracción.
- Legibilidad y Simplicidad del lenguaje.
- Expresiones no Ambiguas.
- Soporte e Integrabilidad.

1.5. Ejemplos de Implementaciones

1.5.1. Umple

Un caso con bastante aceptación dentro del desarrollo de software llevado mediante modelos es la herramienta Umple [11], la cual tiene como objetivo mitigar varios de los inconvenientes enumerados anteriormente, con respecto a la renuencia de los equipos de trabajo para el modelado y al problema que cada desarrollador ve en un desarrollo centrado en la codificación [2], [12], esto a través de la aplicación de refactorizaciones al código lo cual da como resultado un programa equivalente al original, con el agregado de que este puede ser renderizado y editado mediante herramientas UML [13], siguiendo diferentes paradigmas dentro del desarrollo de software tales como el uso de un Lenguaje DSL (`Sección 1.4`).

1.5.2. Telosys

El análisis realizado en [14] lo muestra como una herramienta simple (que utiliza cuestiones tales como los DSL para la creación del modelo), provee la habilidad de la generación de código teniendo como base un modelo, el cual se le provee a la misma mediante una interfáz de línea de comandos, su objetivo es proveer una alternativa al clásico "Primero el UML" dentro del desarrollo, esto significa, que en vez de invertir el tiempo del desarrollador al inicio del proyecto documentando diagramas UML, teniendo como principio lo que se expuso en la `Sección 1.3` se tiene que el modelo **es** el código, es decir, que los límites que separan la documentación de la codificación se desvanecen.

Capítulo 2

Director

2.1. Problematica

Descripcion del problema que se pretende mitigar con el lenguaje

El presente trabajo pretende presentar una herramienta que aplique los conceptos vistos en la `Sección 1.2` referentes a MDSD, focalizando el uso de modelos UML, tales restricciones determinan que se siguen los patrones relacionados a MDA como se acotó en la `Sección 1.3`, como se menciona en la sección para MSDS, para la implementación de tales técnicas, uno de los requisitos es la definición de un Lenguaje Específico de Dominio, (tratado en la `Sección 1.4`).

Director, proveera de un lenguaje especifico de dominio que permite el modelado de sistemas de información orientados a objetos, en esta primera iteración del mismo, se pretende incluir las entidades utilizadas en la realización de gráficas de Diagramas de Clase pertenecientes a la familia de Diagramas UML.

Estructura del Lenguaje

A continuación, se presentarán en diferentes secciones los componentes que hacen al lenguaje, estos se ubicaran según el nivel de dependencia de cada uno con respecto a los demás componentes, de esta manera, compartimentando la responsabilidad se hace a una explicación mas simplificada de la herramienta.

Se pueden esperar, para cada sección que describe a un elemento del lenguaje, las siguientes notaciones que formalizan a la propuesta del presente documento: `Gramática libre de contexto en formato BNF (Backus-Naur)`, `Expresiones Regulares`, `Autómatas Finitos`.

2.2. Palabras Reservadas

Aquí se describirán las palabras utilizadas como tokens en el lenguaje propuesto, estas permiten, desde el establecimiento de relaciones entre entidades

de un modelo, hasta la habilidad de indicar el tipo de devolución de un método. A continuación se proporciona una lista de palabras reservadas con su correspondiente explicación:

class Esta palabra se utiliza como token para la definición de una clase, es un componente de alto nivel dentro del lenguaje.

Ejemplo:

```
class Persona {
  ...
  }
```

relationships Palabra utilizada para establecer relaciones inter-clase dentro de un modelo, esta es una entidad de alto nivel en el lenguaje que permite la relación mediate *herencia*, *asociación*, etc, entre clases de un modelo.

Ejemplo:

```
...
relationships {
  ...
  }
```

void Token que permite establecer el tipo de la devolución de un método, éste indica que el método no hace ninguna devolución.

Ejemplo:

```
...
<visibilidad> metodo():void
...
```

integer Token que permite indicar el tipo de dato devuelto por un metodo, o el tipo de dato de un atributo. Hace referencia al tipo de datos entero; equivalente en un lenguaje como Java, `int`.

Ejemplo:

```
...
<visibilidad> atributo:integer
<visibilidad> metodo():integer
...
```

double Token que permite indicar el tipo de dato devuelto por un metodo, o el tipo de dato de un atributo. Hace referencia al tipo de datos doble; equivalente en un lenguaje como Java, `double`.

Ejemplo:

```
...
<visibilidad> atributo:double
<visibilidad> metodo():double
...
```

float Token que permite indicar el tipo de dato devuelto por un metodo, o el tipo de dato de un atributo. Hace referencia al tipo de datos con presición flotante; equivalente en un lenguaje como Java, `float`.

Ejemplo:

```
...
<visibilidad> atributo:float
<visibilidad> metodo():float
...
```

long Token que permite indicar el tipo de dato devuelto por un metodo, o el tipo de dato de un atributo. Hace referencia al tipo de datos largo; equivalente en un lenguaje como Java, `long`.

Ejemplo:

```
...
<visibilidad> atributo:long
<visibilidad> metodo():long
...
```

boolean Token que permite indicar el tipo de dato devuelto por un metodo, o el tipo de dato de un atributo. Hace referencia al tipo de datos booleanos; equivalente en un lenguaje como Java, `boolean`.

Ejemplo:

```
...
<visibilidad> atributo:boolean
<visibilidad> metodo():boolean
...
```

string Token que permite indicar el tipo de dato devuelto por un metodo, o el tipo de dato de un atributo. Hace referencia al tipo de datos de cadena de caracteres ; equivalente en un lenguaje como Java, `String`; equivalente en C, `char[N]`.

Ejemplo:

```
...
<visibilidad> atributo:string
<visibilidad> metodo():string
...
```

char Token que permite indicar el tipo de dato devuelto por un metodo, o el tipo de dato de un atributo. Hace referencia al tipo de datos de caracter; equivalente en un lenguaje como Java, `char`.

Ejemplo:

```
...
<visibilidad> atributo:char
<visibilidad> metodo():char
...
```

list Token que permite indicar el tipo de dato devuelto por un metodo, o el tipo de dato de un atributo. Hace referencia a una estructura de colección reminiscente a las listas; equivalente en un lenguaje como Java, `List`.

Ejemplo:

```
...
<visibilidad> atributo:list<tipo>
<visibilidad> metodo():list<tipo>
...
```

set Token que permite indicar el tipo de dato devuelto por un metodo, o el tipo de dato de un atributo. Hace referencia a una estructura de colección reminiscente a los conjuntos; equivalente en un lenguaje como Java, `Set`.

Ejemplo:

```
...
<visibilidad> atributo:set<tipo>
<visibilidad> metodo():set<tipo>
...
```

public Esta palabra permite establecer la visibilidad de un elemento dentro del modelo, este elemento puede ser un atributo o una clase.

Ejemplo:

```
...
public atributo:<tipo>
public metodo():<tipo>
...
```

private Esta palabra permite establecer la visibilidad de un elemento dentro del modelo, este elemento puede ser un atributo o una clase.

Ejemplo:

```
...
private atributo:<tipo>
private metodo():<tipo>
...
```

protected Esta palabra permite establecer la visibilidad de un elemento dentro del modelo, este elemento puede ser un atributo o una clase.

Ejemplo:

```
...
protected atributo:<tipo>
protected metodo():<tipo>
...
```

derivate Esta palabra permite establecer la visibilidad de un elemento dentro del modelo, este elemento puede ser un atributo o una clase.

Ejemplo:

```
...
derivate atributo:<tipo>
derivate metodo():<tipo>
...
```

package Esta palabra permite establecer la visibilidad de un elemento dentro del modelo, este elemento puede ser un atributo o una clase.

Ejemplo:

```
...
package atributo:<tipo>
package metodo():<tipo>
...
```

@id Esta palabra se utiliza como un modificador en el ambito de los atributos de clase, este permite brindar información extra acerca del atributo en cuestión, en específico, este indica que el atributo es parte del identificador de la clase que lo posee.

Ejemplo:

```
...
<visibilidad> atributo:<tipo>{@id}
...
```

@readOnly Esta palabra se utiliza como un modificador en el ambito de los atributos de clase, este permite brindar información extra acerca del atributo en cuestión, en específico, este indica que el atributo solo podra ser accedido para lectura una vez se haya guardado.

Ejemplo:

```
...
<visibilidad> atributo:<tipo>{@readOnly}
...
```

@sequence Esta palabra se utiliza como un modificador en el ambito de los atributos de clase, este permite brindar información extra acerca del atributo en cuestión, en específico, este indica que el valor que tome el atributo sigue una secuencia.

Ejemplo:

```
...
<visibilidad> atributo:<tipo>{@sequence}
...
```

@unique Esta palabra se utiliza como un modificador en el ambito de los atributos de clase, este permite brindar información extra acerca del atributo en cuestión, en específico, este indica que el atributo no tiene duplicados.

Ejemplo:

```
...
<visibilidad> atributo:<tipo>{@unique}
...
```

2.3. Símbolos Especiales

En Director, varios de los simbolos especiales permiten hacer uso de las confección gráficas de UML para la confección del modelo, por ejemplo, se tienen varios símbolos que tienen una contraparte en las palabras reservadas para la descripcion de algun método/atributo. En el `Cuadro 2.1` se presentan los símbolos que componen al lenguaje y una breve explicación de cada uno de ellos.

Cuadro 2.1: Símbolos especiales

Símbolo	Descripción
+	Visibilidad de un método/atributo, equivalente a public
-	Visibilidad de un método/atributo, equivalente a private
#	Visibilidad de un método/atributo, equivalente a protected
/	Visibilidad de un método/atributo, equivalente a derivate
∼	Visibilidad de un método/atributo, equivalente a package
:	Permite la asignación de un tipo de dato a un método/atributo
##	Comentarios de línea dentro del modelo
#{	Inicio comentarios multilínea dentro del modelo
}#	Fin comentarios multilínea dentro del modelo
#+	Definición de metainformación para el modelo
//	Comentario de línea para el código resultante del modelo
/*	Inicio comentarios multilínea para el codigo resultante del modelo
*/	Fin comentarios multilínea para el codigo resultante del modelo

2.4. Definición de componentes comunes

En esta sección se pretende ubicar a los bloques que hacen a la base del lenguaje, cuestiones que se utilizan como herramientas para definir otros componentes mas complejos.

2.4.1. letra

Segun la notación BNF se tiene lo siguiente:

```
<letra> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n |
o | p | q | r | s | t | u | v | w | x | y | z | A | B | C | D |
E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T |
U | V | W | X | Y | Z
```

y para un patrón para el motor de expresiones regulares se tiene lo siguiente para las letras:

```
Regex: /[a-zA-Z]/
```

2.4.2. digito

La notación para la gramática libre de contexto en formato BNF para los `dígitos` es la siguiente:

```
<digito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

En cuanto a la expresión regular para los mismos se define de la siguiente manera:

```
Regex: /[0-9]/
```

2.4.3. simbolo

BNF para describir un simbolo dentro del lenguaje se expone a continuación:

```
<simbolo> ::= "|" | " " | "!" | "#" / "$" / "%" / "&" / "(" / ")"
| "*" | "+" | "," | "-" | "." | "/" | ":" | ";" | ">" | "=" | "<"
| "?" | "@" | "[" | "\" | "]" | "^" | "_" | "`" | "{" | "}" | "~"
```

2.4.4. texto

La definición de este elemento simplifica futuro manejo de cuestiones dentro de la definición del lenguaje, el elemento `texto` es la aplicación sucesiva de los elementos `letra`, `digito` y `simbolo`.

El BNF para el elemento se describe de la siguiente manera:

```
<texto> ::= <letra|digito|simbolo> <texto>
```

El autómata finito para este explica al relación de los tres elementos mencionados:

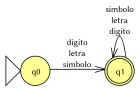

Figura 2.1: Automata finito para texto

2.4.5. nombre

Esta expresión se utiliza a lo largo del documento para hacer referencia a cadenas de texto que identifiquen un elemento del lenguaje, como por ejemplo, utilizando con la palabra `class` este pasaría a definir el nombre de la clase, si se utiliza con en un contexto de atributo, este pasaría a formar parte del nombre que identifica a ese atributo, a continuación se presenta la gramática libre de contexto y la expresión regular para un elemento `nombre` dentro del lenguaje:

Primero se realiza la definicion de una herrameinta que nos permita el uso recursivo de `letra`, `simbolo` y `digito`. Si bien esto se parece bastante a lo que se expone en la `Subsección 2.4.4` per se tiene la necesidad de restringir el uso de símbolos a unicamente el uso del guión bajo "_".

Fragmento 2.1: BNF intermedio definición nombre

```
<nombre-texto> ::= <letra|digito|"_"> <nombre-texto>
```

Fragmento 2.2: BNF para un nombre

```
<nombre> ::= <letra> <nombre-texto>
```

En cuanto la expresión regular para la expresión anterior:

Fragmento 2.3: regex para un nombre

```
Regex: /[a-zA-Z][a-zA-Z0-9]*/
```

2.4.6. visibilidad

La visibilidad, como ya se explicó implicitamente en la descripción de algunas de las palabras reservadas expuestas en la `Sección 2.2`, tiene la función de establecer que miembros del modelo tienen acceso ciertos elementos, el hecho de que estas esten compuestas de palabras reservadas hace que su generalización con componentes de mas bajo nivel descritos en las `Secciones 2.4.1`, `2.4.2` y

2.4.5 no sea posible, por esta razón se hace la descripción con los componentes literales que hacen a la visibilidad.

La notacion BNF para la visibilidad es como se expone a continuación:

```
<visibilidad> ::= public | private | protected | derivate | package
```

Si bien esto permiten describir la visibilidad de los componentes del modelo, según la especificación UML, en su totalidad, en el apartado 2.3 expone que el lenguaje preserva la naturaleza gráfica de UML permitiendo que el usuario utilice simbolos para la definición de visibilidad en diferentes componentes de un modelo, por lo tanto, teniendo esto en consideración, la nueva expresión BNF para la visibilidad es como sigue:

```
<visibilidad> ::= public | + | private | - | protected | # |
derivate | / | package | ~
```

También se presenta la expresion regular correspondiente a la notatión presentada anteriormente:

```
Regex: /(public|+|private|-|protected|#|derivate|/|package|~)/
```

2.4.7. tipo

Este componente permite identificar la naturaleza de metodos y atributos dentro de un modelo, estos, al igual que los componentes de la visibilidad explicados en la Subsección 2.4.6, están basados en literales que forman parte del conjunto de palabras reservadas del lenguaje, esta es la razon por la cual el tipo tampoco se puede generalizar como también es mencionado en la subsección anterior.

El tipo se puede describir de la siguiente manera:

```
<tipo> ::= integer | double | float | long | boolean |
string | char | list<<tipo>> | set<<tipo>>
```

en cuanto a la expresión regular que concierne a la gramática anterior se describe a continuación como:

```
Regex: //
```

2.5. Comentario

De vez en cuando, no solamente el código generado va a necesitar una breve explicación acerca de cual es su función, sino que puede llegar a ser el caso

que el modelo en sí tambien necesite una explicación sobre alguna parte en especial para que los involucrados con el modelo puedan saber de que se trata. El lenguaje propuesto permite este comportamiento mediante la inclusión, no solamente de un mecanismo que permita comentar el código resultante, sino que tambien se puede comentar partes del modelo que no se deben tener en cuenta.

2.5.1. Comentarios Director

Aqui se detallan las reglas a seguir para poder realizar un comentario en el modelo, es decir, este comentario no sería tomado en cuenta a la hora de parsear el archivo para poder obtener el código.

Dentro del área de los comentarios en el lenguaje se ofrece, al igual que gran parte de los lenguajes de propósito general, la posibilidad de introducir comentarios de linea y comentarios multilínea.

La expresión para el BNF de un comentario tanto de línea como uno multilínea está dada de la siguiente manera:

Fragmento 2.4: BNF para comentario de línea

```
<comentario-linea> ::= "##" <texto>
```

Fragmento 2.5: BNF para comentario multilínea

```
<comentario-multilinea> ::= "#{" <texto> "}#"
```

Las respectivas expresiones regulares para ambos tipos de comentarios se describen a continuación:

Fragmento 2.6: regex para comentario de línea

```
regex: /##[a-za-z0-9 \n\r]*/
```

Fragmento 2.7: regex para comentario multilínea

```
regex: /#{[a-za-z0-9 \n\r]}#*/
```

Autómata finito

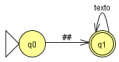

Figura 2.2: Autómata finito para comentario de línea

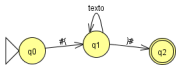

Figura 2.3: Autómata finito para comentario multilínea

2.5.2. Comentarios para Lenguaje

Estos comentarios se tendrán en cuenta a la hora de parsear el documento esto se debe a que serán contenidos por el código resultante. Es decir, son un comentario para el codigo que resulte del módelo que se tenga en cuestión.

Nuevamente, aquí se tiene la posibilidad de establecer los comentarios de línea y comentarios multilínea,

Fragmento 2.8: BNF para comentario de línea del lenguaje

```
<comentario-linea> ::= "//" <texto>
```

Fragmento 2.9: BNF para comentario multilínea del lenguaje

```
<comentario-multilinea> ::= "/*" <texto> "*/"
```

Las respectivas expresiones regulares para ambos tipos de comentarios se describen a continuación:

Fragmento 2.10: regex para comentario de línea del lenguaje

```
regex: /\/\/[a-za-z0-9 \n\r\t]*/
```

Fragmento 2.11: regex para comentario multilínea del lenguaje

```
regex: /\/\*[a-za-z0-9 \n\r\t]*\*\//
```

Autómata finito

Figura 2.4: Autómata finito para comentario de línea del lenguaje

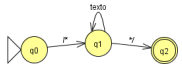

Figura 2.5: Autómata finito para comentario multilínea del lenguaje

2.6. Atributo

Los atributos son los elementos utilizados en el standard UML para designar propiedades que son inherentes de clases, una vez instanciados estos toman valor y ayudan al manejo de datos del objeto, el lenguaje propuesto permite el manejo de los atributos y algunas otras cuestiones que están relacionadas con estos.

La definición de un atributo según notacion BNF es la siguiente:

```
<atributo> ::= <visibilidad> <espacio-blanco> <nombre> ":" <tipo>
```

Teniendo en cuenta que se tiene que estar en el contexto de una clase para que el mismo tenga sentido, a continuación un ejemplo:

`Consigna`: *se desea tener una clase que permita el manejo de información de una persona, la información a manejar es el nombre, apellido y DNI.*

teniendo esta información se puede armar un modelo en el lenguaje director que concuerde con estos requerimientos. Dado a que por el momento unicamente se está tratando con los atributos no se tendrá en cuenta la declaración de la clase `Persona` mencionada en la consigna propuesta.

```
...
  private nombre:string
  private apellido:string
  private DNI:string
...
```

Lo expuesto en el `Fragmento 2.6`, si bien, demuestra la capacidad del lenguaje de describir una serie de atributos con sus respectivos metadatos (visibilidad y tipo), también se pretende dar los primeros pasos para la futura implementación de la compatibilización con los futuras funcionalidades extraídas de un acercamiento al Lenguaje de Restricción de Objetos, puesto a que la adición de funcionalidades de éstas características automatizaria aún mas el proceso de desarrollo y permitiría el pase de modelo-a-código de manera mas sencilla. Las características que se implementan compatibles funcionalidades brindadas por OCL son las de modificadores para los atributos, ademas de los mencionados anteriormente, estos modificadores ya se mencionaron en la `Sección 2.2`.

Como ejemplo, se podrían aplicar algunas de estas cuestiones a lo implementado al `Fragmento 2.6`.

```
...
  private nombre:string
  private apellido:string
  private DNI:string {@id, @readOnly, @unique}
...
```

De esta forma, lo implementado en el `Fragmento 2.6` brinda información como para que se pueda determinar que el atributo DNI:

- debe ser tratado como un identificador de la clase.
- debe ser de solo lectura, de este modo se deduce que no es necesaria la implementacion de un método `set` para el mismo.
- debe ser unico en el listado de personas, es decir, no se puede repetir.

Lo cual da información relevante para que la herramienta pueda brindar resultados mas cercanos al dominio en el que se esté trabajando.

Se puede dejar expresado que el BNF, teniendo en cuenta la incorporación de lo demostrado quedaría de la siguiente manera:

```
<atributo> ::= <visibilidad> <nombre> ‘‘:’’ <tipo>
‘‘{’’ <modificador> ‘‘}’’
```

```
<modificador> ::= @id | @readOnly | @unique | @sequence | <modificador>
```

2.6.1. Autómata Finito

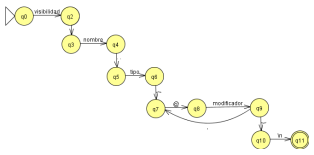

Figura 2.6: Autómata finito para un atributo con restricciones de objeto

2.7. Método

Al igual que los atributos, los metodos, son componentes dentro del diagrama de clases que ayudan a la manipulación de los datos, estos brindan el comportamiento que es propio al objeto. El lenguaje presentado permite definir métodos de la siguiente manera:

```
<metodo>::=<visibilidad><nombre>"("<parametro>")" ":"<tipo>
<metodo>::=<visibilidad><nombre>"("<parametros>")" ":"<tipo>
```

En donde se puede decir que `parametro(s)` se define de la siguiente manera:

```
<parametro> ::= <nombre> ":" <tipo>
```

```
<parametros> ::= <parametro> | <parametros>
```

2.8. Clase

2.9. Relación

Referencias

[1] C. Michel R. V., "Empirical Studies into UML in Practice: Pitfalls and Prospects", *IEEE/ACM 9th International Workshop on Modelling in Software Engineering*, 2017. DOI: 10.1109/MiSE.2017.24.

[2] A. Aldaeej y O. Badreddin, "Towards Promoting Design and UML Modelling Practices in the Open Source Community", *IEEE/ACM 38th International Conference on Software Engineering Companion*, 2016. DOI: 10.1145/2889160.2892649.

[3] A. Forward y T. Lethbridge, "Perceptions of Software Modeling: a Survey of Software Practitioners", *5th Workshop from Code Centric to Model Centric: Evaluating the Effectiveness of MDD*, 2010.

[4] B. Rumpe, *Modelling With UML, Language, Concepts, Methods*. Aachen, Alemania: Springer, 2004. DOI: 10.1007/978-3-319-33933-7.

[5] B. Anda, K. Hansen, I. Gullesen y H. K. Thorsen, "Experience from Introducing UML-based Development in a Large Safety-Critical Project", *Empirical Software Engineering*, 2006. DOI: 10.1007/s10664-006-9020-6.

[6] T. Stahl y M. Völter, *Model-Driven Software Development, Technology, Engineering, Management*. England: John Wiley y Sons, Ltd, 2006, págs. 3-27.

[7] J. D. Poole, "Model Driven Architecture: Vision, Standards and Emerging Technologies", *ECOOP, Workshop on Metamodeling and Adaptative Object Models*, 2001.

[8] S. Sobernig, B. Hoisl y M. Strembeck, "Extracting Reusable Design Desicions for UML-Based Domain-Specific Languages: A Multi-Method Study", *Journal of Systems and Software*, 2016. DOI: 10.1016/j.jss.2015.11.037.

[9] V. C. Nguyen, X. Qafmolla y K. Richta, "Domain Specific Language Approach on Model-Driven Development of Web Services", *Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University*, vol. 11, n.º 8, 2014.

[10] M. Mazanec y M. Ondřej, "On General-Purpose Textual Modeling Languages", *Department of Computer Science, FEL, Czech Technical University*, págs. 1-12, 2012.

[11] U. Team. (2018). Umple, dirección: http://cruise.site.uottawa.ca/umple/.

[12] M. A. Garzón, H. Aljamaan, T. C. Lethbridge y O. Bradeddin, "Reverse Engineering of Object-Oriented Code into Umple using an incremental and Rule-Based Approach", 2014.

[13] T. C. Lethbridge, A. Forward y O. Badreddin, "Umplification: Refactoring to Incrementally add Abstraction to a Program", *Working Conference on Reverse Engineering*, vol. 17, 2010.

[14] L. Guérin. (2018). Telosys: the Concept of Lightweight Model for Code Generation, dirección: https://modeling-languages.com/telosys-tools-the-concept-of-lightweight-model-for-code-generation/.