

DIRECTOR
Teoría de la Computación

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

01 de Noviembre, 2018

Universidad Nacional de Misiones

Facultad de Ciencias Exactas Químicas y
Naturales

Teoría de la Computación
Trabajo Integrador

Definición de un Lenguaje Específico de
Dominio para la Generación de Código
partiendo de un Modelo UML: Director

Autores:

R. C. Ulises [LU: LS00704]
C. J. Héctor [LU: LS00353]

Año: 2018

Versionado

Se mantiene un versionado del presente documento con el fin de mantener un respaldo del trabajo, y además proveer a la cátedra, o cualquier interesado, la posibilidad de leer el material en la última versión disponible.

REPOSITORIO: *<https://github.com/ulisescolina/UC-TC>*

–EQUIPO DIRECTOR

Índice general

1. Estado de la Cuestión	1
1.1. Introducción	1
1.2. Desarrollo de Software Dirigido por Modelos	2
1.3. Arquitectura Dirigida por Modelos	3
1.4. Lenguaje Específico de Dominio	3
1.5. Ejemplos de Implementaciones	4
1.5.1. Umple	4
1.5.2. Telosys	4
2. Director	5
2.1. Problemática	5
2.2. Palabras Reservadas	5
2.3. Símbolos Especiales	10
2.4. Definición de componentes comunes	11
2.4.1. letra	11
2.4.2. digito	11
2.4.3. simbolo	12
2.4.4. texto	12
2.4.5. nombre	12
2.4.6. visibilidad	13
2.4.7. tipo	14
2.5. Comentario	14
2.5.1. Comentarios Director	14
2.5.2. Comentarios para Lenguaje	16
2.6. Atributo	17
2.7. Método	19
2.8. Clase	21
2.9. Relación	23
2.10. Metadatos	26
3. Anexos	28
3.1. Ejemplo 1	28
3.2. Ejemplo 2	30
3.3. Ejemplo 3	36

Índice de figuras

2.1. Autómata finito - Texto	12
2.2. Autómata finito - Nombre	13
2.3. Autómata finito - Comentario de línea	15
2.4. Autómata finito - Comentario multilínea	15
2.5. Autómata finito - Comentario de línea (lenguaje)	16
2.6. Autómata finito - Comentario multilínea (lenguaje)	16
2.7. Autómata finito - Atributo con restricciones de objeto	17
2.8. Diagrama de Clase - representación de la consigna	17
2.9. Autómata finito - Modificador para los atributos	19
2.10. Autómata finito - Método	21
2.11. Autómata finito - Parámetro	21
2.12. Autómata finito - Clase	23
2.13. Autómata finito - Relationships	23
2.14. Autómata finito - Relación	24
2.15. Autómata finito - Cardinalidad para una Relación	25
2.16. Autómata finito - Tipos de Relación	25
3.1. Diagrama de Clases - Modelo ejemplo 1	28
3.2. Diagrama de Clases - Modelo ejemplo 2	31
3.3. Diagrama de Clases - Modelo ejemplo 3	37

Índice de Fragmentos de Código

2.1. BNF - Comentario de línea	15
2.2. BNF - Comentario multilínea	15
2.3. Regex - Comentario de Línea	15
2.4. Regex - Comentario Multilínea	15
2.5. BNF - Comentario de línea (lenguaje)	16
2.6. BNF - Comentario multilínea (lenguaje)	16
2.7. Regex - Comentario de línea (lenguaje)	16
2.8. Regex - Comentario multilínea (lenguaje)	16
2.9. BNF - Atributo	17
2.10. Director - Modelado de atributos	18
2.11. Director - Modelao de atributos (con OCL)	18
2.12. BNF - Atributo	19
2.13. BNF - Método	19
2.14. Java - Generación Fragmento 2.6	20
2.15. Director - Declaración de Método	20
2.16. Java - Generación metodo_generico agregado en Fragmento 2.15	21
2.17. BNF - Clase	21
2.18. Director - Adhiere componente clase al ejemplo	22
2.19. Java - Resultado obtenido de la generación de Fragmento 2.18	22
2.20. BNF - Relationships	23
2.21. BNF - Relación	24
2.22. BNF - Rol	24
2.23. BNF - Cardinalidad para una Relación	25
2.24. BNF - Tipos de Relación	25
2.25. Director - Inicio y Fin del Modelo	26
2.26. Director - Metadatos ejemplos	26
3.1. Director - Modelo para la Figura 3.1	29
3.2. Python - Ejemplo 1, Código generado	29
3.3. Director - Modelo equivalente a la Figura 3.2	31
3.4. Java - Ejemplo 2, Clase generada Facturacion.java	32
3.5. Java - Ejemplo 2, Clase generada TipoFactura.java	32
3.6. Java - Ejemplo 2, Clase generada Producto.java	33

3.7. Java - Ejemplo 2, Clase generada <code>DescripcionProducto.java</code> .	33
3.8. Java - Ejemplo 2, Clase generada <code>Factura.java</code>	34
3.9. Director - Modelo equivalente a la Figura 3.3	37
3.10. Java - Ejemplo 3, Clase generada <code>Reparaciones.java</code>	38
3.11. Java - Ejemplo 3, Clase generada <code>Reclamo.java</code>	38
3.12. Java - Ejemplo 3, Clase generada <code>Articulo.java</code>	40
3.13. Java - Ejemplo 3, Clase generada <code>TareaDefinida.java</code>	41
3.14. Java - Ejemplo 3, Clase generada <code>TipoDeArticulo.java</code>	42
3.15. Java - Ejemplo 3, Clase generada <code>TareaARealizar.java</code>	43
3.16. Java - Ejemplo 3, Clase generada <code>TiempoInvertido.java</code>	45
3.17. Java - Ejemplo 3, Clase generada <code>Tecnico.java</code>	45
3.18. Java - Ejemplo 3, Clase generada <code>EmpleadoMensual.java</code>	46
3.19. Java - Ejemplo 3, Clase generada <code>EmpleadoJornalero.java</code>	48

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 década de los 1990 siendo la idea principal del mismo, el modelado de sistemas de software en varios niveles de abstracción [1], además considerándose por estudios como la herramienta de notación para modelado de software dominante [2] aunque usado predominantemente de manera informal y que las actividades relacionadas 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 del resultado de los estudios mencionados, algunas encuestas realizadas en los mismos muestran que los desarrolladores prefieren centrar el trabajo en vastas cantidad de líneas de código, raramente mirando diagramas, y mucho menos confeccionándolos o editándolos, posibles razones para este comportamiento, en principio antagónico, en donde los efectos positivos de UML pueden verse reducidos [5] se describen a continuación: *la no-factibilidad de realizar ingeniería inversa en código 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 razón 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 del agrado del desarrollador*.
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. La reestructuración del proyecto es muy compleja.
5. Los cambios de código pueden producir bugs.

1.2. Desarrollo de Software Dirigido por Modelos

El desarrollo de software dirigido por modelos (MDSD, de ahora en más, 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 también es considerado igual a código; incluso es aplicable en campos de alta especialidad debido a una de las características del acercamiento: realización de abstracciones específicas de dominio y 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 código ejecutable en varias plataformas.

1.3. Arquitectura Dirigida por Modelos

La arquitectura dirigida por modelos (MDA, por sus siglas en inglés) es una iniciativa introducida por el *Object Management Group*, con el propósito 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 núcleo del problema de interoperabilidad lo cual hace que la implementación del sistema sea independiente de la tecnología.

La OMG, promueve MDA como un *marco de trabajo arquitectónico para el desarrollo de software*, el cual está construido alrededor de un número 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 más restrictivo, enfocándose principalmente en lenguajes de modelado basados en UML [6].

1.4. Lenguaje Específico de Dominio

Un lenguaje específico 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 sintaxis y semántica 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 más probable que las personas sepan cómo 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. Problemática

Descripción 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 específico 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 ubicarán 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 más 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 {  
    ...  
}
```

abstract Token que permite designar a un componente como abstracto.

Ejemplo:

```
abstract 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 mediante *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 método, 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 precisión 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 símbolos especiales permiten sacar provecho de las propiedades gráficas de UML para la confección del modelo, por ejemplo, se implementan símbolos que representan una contraparte en las palabras reservadas para la descripción de algún 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/a- tributo
##	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 código resultante del modelo
*/	Fin comentarios multilínea para el código 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 de expresiones regulares se tiene lo siguiente con respecto a 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:

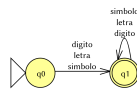


Figura 2.1: Autómata finito - Texto

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 sería el 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, antes, se crea una herramienta que 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** pero se tiene la necesidad de restringir el uso de símbolos a unicamente el uso del guión bajo “_”.

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

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

En cuanto la expresión regular para la expresión anterior:

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

El autómata finito para este elemento es el siguiente:



Figura 2.2: Autómata finito - Nombre

2.4.6. visibilidad

La visibilidad, como ya se explicó implícitamente 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 estén 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 preserve 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 expresión regular correspondiente a la notación presentada anteriormente:

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

2.4.7. tipo

Este componente permite identificar la naturaleza de métodos 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 razón 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 cuál es su función, sino que puede llegar a ser el caso que el modelo en sí también necesite una explicación sobre alguna parte en especial, para que los involucrados con el modelo puedan saber de qué se trata. El lenguaje propuesto permite este comportamiento mediante la inclusión, no solamente de un mecanismo que permita comentar el código resultante, sino que también se puede comentar partes del modelo que no se deben tener en cuenta.

2.5.1. Comentarios Director

Aquí 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 línea y comentarios multilínea.

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

Fragmento 2.1: BNF - Comentario de línea

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

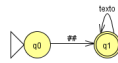


Figura 2.3: Autómata finito - Comentario de línea

Fragmento 2.2: BNF - Comentario multilínea

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

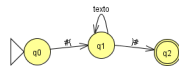


Figura 2.4: Autómata finito - Comentario multilínea

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

Fragmento 2.3: Regex - Comentario de Línea

```
regex: /##[a-zA-Z0-9 \n\r\t]*/
```

Fragmento 2.4: Regex - Comentario Multilínea

```
regex: /#{[a-zA-Z0-9 \n\r\t]}#*/
```

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 código que resulte del modelo que se tenga en cuestión.

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

Fragmento 2.5: BNF - Comentario de línea (lenguaje)

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

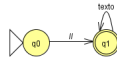


Figura 2.5: Autómata finito - Comentario de línea (lenguaje)

Fragmento 2.6: BNF - Comentario multilínea (lenguaje)

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

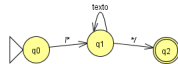


Figura 2.6: Autómata finito - Comentario multilínea (lenguaje)

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

Fragmento 2.7: Regex - Comentario de línea (lenguaje)

```
regex: /\[/[a-zA-Z0-9 \n\r\t]*\//
```

Fragmento 2.8: Regex - Comentario multilínea (lenguaje)

```
regex: /\[/\*[a-zA-Z0-9 \n\r\t]*\*\//
```

2.6. Atributo

Los atributos son los elementos utilizados en el standard UML para designar propiedades que son inherentes a las clases, una vez instanciadas (las clases) estos (los atributos) 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:

Fragmento 2.9: BNF - Atributo

```
<atributo> ::= <visibilidad> <nombre> ":" <tipo>
```

Este elemento dentro del lenguaje se puede ver en un gráfico en un autómata finito de la siguiente manera:

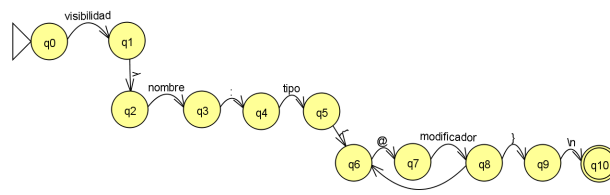


Figura 2.7: Autómata finito - Atributo con restricciones de objeto

Teniendo en cuenta que el atributo tiene que estar definido dentro de otra estructura del estándar UML, una clase.

A continuación un ejemplo, para expresar medianamente el funcionamiento del lenguaje:

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.

Lo cual se puede representar mediante un diagrama de clase de la siguiente manera:

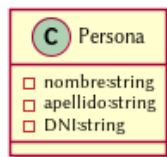


Figura 2.8: Diagrama de Clase - representación de la consigna

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.*

Fragmento 2.10: Director - Modelado de atributos

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

Lo expuesto en el **Fragmento 2.10**, demuestra la capacidad del lenguaje de describir una serie de atributos con sus respectivos metadatos (visibilidad y tipo), sin embargo, cabe destacar que también se deja abierta la posibilidad de la futura implementación y la compatibilización con funcionalidades relacionadas al Lenguaje de Restricción de Objetos, puesto a que la adición de éstas características automatizaría aún más 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, estos modificadores ya se mencionaron en la **Sección 2.2**.

Como ejemplo, se podrían aplicar algunas de estas cuestiones a lo implementado en **Fragmento 2.10** resultando lo siguiente:

Fragmento 2.11: Director - Modelao de atributos (con OCL)

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

De esta forma, lo implementado en el **Fragmento 2.11** 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.

Con esto, los resultados de la herramienta hacen que lo generado esté aun más cerca 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:

Fragmento 2.12: BNF - Atributo

```
<atributo> ::= <visibilidad> <nombre> ":" <tipo>
              "{" <modificadores> "}"
```

En donde **modificadores** es la pluralización de un **modificador**:

```
<modificadores> ::= <modificador> | <modificadores>
```

y **modificador** se define de la siguiente manera:

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

En donde, además, se puede definir un autómata para los modificadores que brindan las restricciones para el objeto, de la siguiente manera:

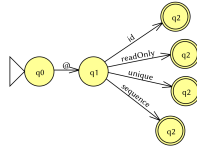


Figura 2.9: Autómata finito - Modificador para los atributos

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:

Fragmento 2.13: BNF - Método

```
<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>
```

Director, está dotado de la capacidad para tratar los casos triviales del manejo de los datos dentro de un objeto, estos casos triviales corresponden a situaciones en las cuales una de las necesidades de comportamiento para un objeto

viene dado por la urgencia de agregar algún mecanismo que permita acceder a los atributos del mismo, las acciones comunes de acceso a estos atributos son las siguientes: definir el valor de un atributo y obtener el valor de ese atributo, las cuales son solucionadas con los **setters** y los **getters**.

Director, no requiere que se definan explícitamente estos métodos, éste los genera de manera automática.

Teniendo en cuenta el ejemplo mencionado en la **Sección 2.6**, Director automáticamente generaría los siguientes métodos para el lenguaje Java:

Fragmento 2.14: Java - Generación **Fragmento 2.6**

```
...
public String get_nombre(){
    return this.nombre;
}

public void set_nombre(String nombre){
    this.nombre = nombre;
}

public String get_apellido(){
    return this.apellido;
}

public void set_apellido(String apellido){
    this.apellido = apellido;
}

public String get_DNI(){
    return this.DNI;
}

public void set_DNI(String DNI){
    this.DNI = DNI;
}
...
```

De todos modos, es posible indicar otros métodos que se deseen incluir en el modelo. Si agregáramos algún *metodo_generico* al ejemplo con el que se viene trabajando, la lista de atributos y modelos quedaría definida de la siguiente manera.

Fragmento 2.15: Director - Declaración de Método

```
private nombre:string
private apellido:string
private DNI:string
public metodo_generico():void
```

Lo que resultaría en la generación del “esqueleto” necesario para la implementación de dicho método. Tomando como base lo que ya se hizo en el **Fragmento 2.14**, se le agregaría:

Fragmento 2.16: Java - Generación metodo_generico agregado en Fragmento 2.15

```
public void metodo_generico() {
    // Implementacion del metodo
}
```

Autómatas finitos

Aquí se proporcionarán los autómatas finitos para la definición de un método y las subcomponentes que se crearon para éste, como por ejemplo el **parametro**.

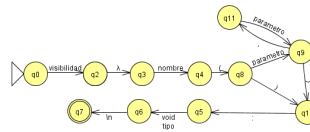


Figura 2.10: Autómata finito - Método

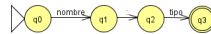


Figura 2.11: Autómata finito - Parámetro

2.8. Clase

Hasta ahora se fueron definiendo componentes básicos que pertenecen al estándar UML, específicamente, subcomponentes los cuales definen, como ya se mencionó, los datos y el comportamiento de los objetos instanciados, ahora, es necesaria otra entidad que permita la conjunción de estas herramientas, para eso, se define el componente de **clase**.

La notación BNF para la definición de una clase en Director es la siguiente:

Fragmento 2.17: BNF - Clase

```
<clase> ::= "class" <nombre> "{" <atributos|metodos> "}"
```

Los subcomponentes se trataron en secciones anteriores del documento, de todas formas se detalla el BNF correspondiente a la pluralización de los mismos.

```
<atributos> ::= <atributo> | <atributos>
```

```
<metodos> ::= <metodo> | <metodos>
```

Volviendo al ejemplo tratado en la **Sección 2.6**, agregando los que se dijo en el **Fragmento 2.15**, se puede completar la consigna juntando los componentes del lenguaje definidos y ejemplificados anteriormente junto con el elemento que se esta definiendo aquí en un modelo.

Fragmento 2.18: Director - Adhiere componente clase al ejemplo

```
class Persona {  
    private nombre:string  
    private apellido:string  
    private DNI:string {@readOnly, @unique}  
    public metodo_generico():void  
}
```

Lo cual generaría el siguiente código en un lenguaje como Java:

Fragmento 2.19: Java - Resultado obtenido de la generación de **Fragmento 2.18**

```
class Persona{  
    private String nombre;  
    private String apellido;  
    private String DNI;  
  
    public String get_nombre(){  
        return this.nombre;  
    }  
  
    public void set_nombre(String nombre){  
        this.nombre = nombre;  
    }  
  
    public String get_apellido(){  
        return this.apellido;  
    }  
  
    public void set_apellido(String apellido){  
        this.apellido = apellido;  
    }  
  
    public String get_DNI(){  
        return this.DNI;  
    }  
  
    public void set_DNI(String DNI){
```

```

    this.DNI = DNI;
}

public void metodo_generico() {
    // Implementacion del metodo
}
}

```

Además es posible su representación mediante el siguiente autómata finito:



Figura 2.12: Autómata finito - Clase

2.9. Relación

En las Secciones 2.6, 2.7 y 2.8 se definieron los componentes clave que permiten sistematizar la generación de código para los componentes base de los diagramas de clase, en ésta sección, se describe un componente que permite la asociación de diferentes clases dentro de un modelo, una **Relación**.

Estas relaciones permiten la asociación entre dos clases. Director permite definir una sección en la cual se pueden definir un conjunto de relaciones que luego se tendrán en cuenta a la hora de generar los componentes que se definieron en el modelo.

La notación BNF para las relaciones es la siguiente:

Fragmento 2.20: BNF - Relationships

```

<relationships> ::= "relationships {" <lista-relaciones> "}"

```

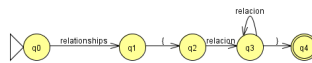


Figura 2.13: Autómata finito - Relationships

en donde **lista-relaciones** se define de la siguiente manera:

```
<lista-relaciones> ::= <relacion> | <lista-relaciones>
```

y además se tiene que **relacion** se define de la siguiente manera:

Fragmento 2.21: BNF - Relación

```
<relacion> ::= <rol-asoc><cardinalidad>  
              <tipo-relacion>  
              <cardinalidad><rol-asoc>
```

Se debe realizar la definicion de **rol-asoc**, **cardinalidad** y **tipo-relacion**.

Definición rol-asoc

Se procede a definir el significado de **rol-asoc**, que lo que representa es el rol que cumple la clase en la relacion a establecerse.

Fragmento 2.22: BNF - Rol

```
<rol-asoc> ::= <nombre-clase>":"<rol>
```

Aquí se tiene que decir que **nombre-clase** hace referencia a una de las clases que se tienen en el modelo para el cual se está describiendo la relación. Luego, se puede definir el significado de **rol** dentro de las relaciones. Conceptualmente, el rol pretende darle un nombre personalizado a la participación de la clase dentro de la relacion, este pasará a formar parte (al momento de la generación del código del modelo) de una de las clases involucradas en la relación a modo de atributo.

Este ultimo (el rol), no es necesario en la definicion de una relación, dado a que si este no se encuentra se le asigna un nombre de rol que consistiría en el nombre de ambas clases involucradas en la relación, en el autómata finito expuesto en la **Figura 2.14** se puede ver esto claramente.



Figura 2.14: Autómata finito - Relación

Definición cardinalidad

Dentro de una relación la cardinalidad representa la cantidad con la que participa la clase en esa asociación con otra clase, de esta manera se pueden representar cuestiones como:

(...) se pretende que el alumno pueda tener muchas materias (...)

Aquí es necesario relacionar dos entidades como ser las del **alumno** y la de **materia** para poder representar lo que se solicita, la cardinalidad en este caso es de **muchos a muchos**, por el hecho de que, además de que el alumno pueda tener muchas materias, a las materias pueden concurrir muchos alumnos, es decir que la materia también puede tener asociada muchos alumnos.

La notación BNF para una relación es la siguiente:

Fragmento 2.23: BNF - Cardinalidad para una Relación

```
<cardinalidad> ::= "<digito|'*> < " | '..' <digito|'*>> "
```

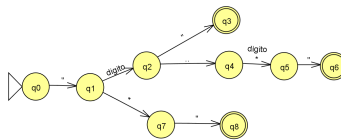


Figura 2.15: Autómata finito - Cardinalidad para una Relación

Definición tipo-relacion

Este elemento, define como se relacionan las clases en una **relacion** dada, por ejemplo, puede ser simplemente de **asociación** o se puede tener una relación de **herencia**.

Director maneja estas cuestiones con componentes que buscan el parecido a la parte gráfica de los diagramas, por este motivo, estos tipos de relación son literales para cada uno de los tipos que se necesite. A continuación se define el BNF para el elemento en cuestión.

Fragmento 2.24: BNF - Tipos de Relación

```
<tipo-relacion> ::= "<|--" | "<|- " | "----" | "-|>" | "-->">
```

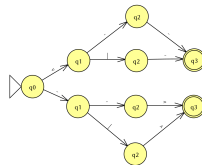


Figura 2.16: Autómata finito - Tipos de Relación

2.10. Metadatos

La naturaleza que rodea a la propuesta, presenta cuestiones que no se pueden solucionar en el modelo dado a que no son inherentes al mismo, para dar una solución a esta situación se decidió la inclusión de otro tipo de elementos al lenguaje, además de los que representan a los elementos utilizados en el diagrama de clases de UML, estos elementos son denominados **Metadatos**.

Los metadatos permiten dar directivas al lenguaje de que se debe realizar con el modelo, ejemplo:

- Lenguaje de salida, este brinda al lenguaje cual debe ser el **lenguaje de salida** del modelo.
- Estructura de salida, este brinda ordenes al lenguaje de si las clases resultantes deben estar separadas en diferentes archivos (obligatorio para lenguajes como Java) o en un solo archivo (Lenguajes como Python permiten esto).

El hecho de que deba existir otro tipo de cuestiones en el archivo que describe al modelo, que no necesariamente tiene que ver con el modelo, hace que también sea necesario agregar algún mecanismo que permita comunicar al lenguaje donde inicia y donde termina el modelo. Esto se realiza con los siguientes literales, y utilizando uno de los simbolos ya mencionados en el **Cuadro 2.1, Sección 2.3**.

Fragmento 2.25: Director - Inicio y Fin del Modelo

```
#+BEGINDRT
...  Descripcion
...    del
...    Modelo
#+ENDDRT
```

Como se aprecia en el **Fragmento 2.25**, la descripción del modelo con todos los elementos descritos anteriormente están envueltos en los literales **BEGINDRT** y **ENDDRT** antecidos por uno de los simbolos especiales **#+**, de la misma forma, los tokens para indicar cuestiones como el lenguaje que se debe generar o la estructura de salida, son precedidos por el mismo simbolo especial, y se conforman de diferentes secciones que permiten la configuración de diferentes aspectos de lo que se obtiene en la salida, se brindará un ejemplo, pero cabe destacar que la descripción de todas las secciones posibles para la configuración no se detalla en el presente documento.

Fragmento 2.26: Director - Metadatos ejemplos

```
1  #+DRT.CONF.O.LANG=java
2  #+JAVA.CONF.O.AUTHOR=Ulices
3  #+JAVA.CONF.F.STRUCT=split
```

A continuación se brinda una explicación de los ejemplos expuestos en el **Fragmento 2.26** primeramente separando los diferentes componentes en áreas de interés, luego comentando lo que cada área pretende representar:

- **Línea 1:** en primer lugar se tiene **DRT**, éste indica el lenguaje para el cual se estará seteando la configuración, el ejemplo concreto indicaría alguna directiva al lenguaje Director de como tratar al modelo, lo siguiente **CONF** indica que se trata de una configuración, luego se tiene **O** el cual se desprende de la palabra *Output* haciendo referencia a la salida del programa y finalmente, se tiene **LANG**, lo cual pasa a ser una propiedad que tomara el valor de 'java' de esta forma se le esta diciendo *“Para lo que tenga que ver con la manipulación de modelos mediante el lenguaje Director, establecer la configuración de lenguaje para la salida del mismo como java”*.
- **Línea 2:** aquí se tiene **JAVA** al principio, por lo tanto lo que sigue será para darle directivas a las salidas que estén en lenguaje Java, luego se tiene **CONF** lo cual indica que se estará tratando de una sección de configuración, después se tiene **O** nuevamente, este viene de output, y finalmente se tiene un atributo para la sección que se denomina **AUTHOR**, éste permitirá a la herramienta establecer el autor que estará trabajando en el código generados.
- **Línea 3:** éste nuevamente inicia con **JAVA** lo que indica que se darán indicaciones que tienen que ver con el lenguaje Java, dentro de este, es de interés la categoría configuración (**CONF**) y aquí, en la configuración interesa la parte ya “física” (por la **F**, derivado de *“File”*), del archivo, ya se trata de un archivo con código generado en su interior, aquí el atributo que se está estableciendo se denomina **STRUCT** haciendo referencia a la estructura con la que se generará el proyecto, en este caso “split” significaría que las clases estarán en archivos separados, una clase por cada archivo.

Capítulo 3

Anexos

En este capítulo se van a explorar las diferentes escenarios los cuales se abordan, con el modelado UML y posteriormente mediante el uso de las capacidades del lenguaje que se propuso en el documento.

Si bien, la etapa de análisis y diseño dentro de un proyecto de software, la cual brinda los materiales para que lo propuesto pueda generar código, suele ser compleja, en este trabajo no se propone realizar una explicación de la realización de este procedimiento, para esto, pueden chequearse las bibliografías vistas en diferentes cátedras (ejemplos pueden verse en [15], [16], [17], [18], [19]).

Además se puede destacar que el hecho de que los componentes vistos se basan específicamente en el diagrama de clases dentro del estándar UML, estos serán los únicos que se presentan a lo largo del capítulo.

3.1. Ejemplo 1

Para el primero de la serie de ejemplos que se verán en el capítulo se propone la demostración de como la herramienta maneja una relación. a continuación se presenta un caso genérico en el cual se tiene que:

Se tienen dos clases A y B, un objeto de la clase A puede tener asociado a muchos objetos de la clase B, y la clase B puede tener asociado unicamente un objeto A.

Visto en un diagrama de clase, se tendría lo siguiente:



Figura 3.1: Diagrama de Clases - Modelo ejemplo 1

Una vez el modelo esté hecho, es relativamente sencillo transmitirlo a su equivalente en formato de texto. A continuación el modelo escrito en lenguaje Director.

Fragmento 3.1: Director - Modelo para la Figura 3.1

```

1  #+DRT.CONF.O.LANG=python #se define el lenguaje de salida
2  #+DRT.CONF.O.PROJ=Ejemplo #se designa un nombre para el proyecto
3  #+PYTHON.CONF.O.STRUCT=join
4  #+BEGINDRT
5      class A {
6          - atr:integer
7          + met():string
8      }
9      class B {
10         - atr:string
11         + met():integer
12     }
13     relationships {
14         A "1" — "0..*" B
15     }
16 #+ENDDRT

```

Las cuestiones a tener en cuenta a la hora de analizar el resultado obtenido en la generación del código a partir del modelo son las siguientes:

- **Fragmento 3.1/Linea 1:** aquí se determina que el lenguaje de salida será Python.
- **Fragmento 3.1/Linea 4:** esta permite al programa conocer la estructura de salida del código, en este caso se va a presentar el código (con todas las clases involucradas) en un solo archivo, mas adelante se ve otro ejemplo utilizando Java, en donde es obligatorio definir unicamente una¹ clase por archivo.

Éste modelo descrito en lenguaje Director dará como resultado el código presentado en el **Fragmento 3.2:**

Fragmento 3.2: Python - Ejemplo 1, Código generado

```

1  class A:
2      def __init__(self):
3          self.atr = null
4          self.rel_A_B = []
5
6      def met():
7          # implementacion del metodo
8          return 0
9
10
11 class B:
12     def __init__(self, a):
13         self.atr = null
14         self.rel_B_A = a

```

¹Siempre y cuando no se esté hablando de una clase embebida

```

15
16     def met():
17         # implementacion del metodo
18         return ""
19
20
21 if __name__ == "__main__":
22     print("Implementar funcionalidad en caso de ejecucion"+
23         " independiente.")

```

Otra cuestión que puede resultar interesante en el código generado anteriormente es, *¿Donde se ve la relación? ¿Qué nombre se designa a la relación si no se establece ningún rol explícitamente en el modelo? (lo que está sucediendo en dicho código).*

La relación de las clases se representan mediante el uso de atributos, y la aparición de los mismos dependerá de la direccionalidad de la relación [20], [21], en este caso, el diagrama muestra que la relación es *bidireccional*, por lo tanto ambas clases deben tener atributos que representen a la otra, con respecto al nombre a utilizar si no se provee el *rol* que cada clase cumple en la relación, Director genera un nombre generico que contiene a las dos clases en cuestion, en este caso genera `rel_A_B` y `rel_B_A`.

3.2. Ejemplo 2

Como segundo ejemplo, se puede plantear algo que se ve a menudo a la hora de trabajar con sistemas de información, se mantendrá este ejemplo simple para así poder abarcar el resultado de la mejor manera.

Consigna:

Considerese un módulo de sistema que se encarga de la facturación de una entidad, en donde una factura tiene información propia como *número y tipo de factura*, en donde el tipo puede tener información propia como una *descripción e información impositiva* y además, cuenta con una lista de productos (el detalle de la factura).

Una posibilidad para un diagrama de clases que permita modelar este comportamiento es el clásico *cabecera-detalle*, descrito a continuación:

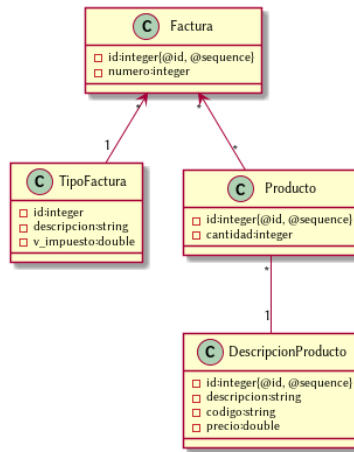


Figura 3.2: Diagrama de Clases - Modelo ejemplo 2

Teniendo este diagrama de clases, se puede transferir la informacion a un modelo Director, éste quedaría de la siguiente manera:

Fragmento 3.3: Director - Modelo equivalente a la Figura 3.2

```

1  #+DRT.CONF.O.LANG=java #se define el lenguaje de salida
2  #+DRT.CONF.O.PROJ=Facturacion #se designa un nombre para el proyecto
3  #+JAVA.CONF.F.STRUCT=split #se define la estructura de salida
4
5  #+BEGINDRT # token inicio de modelo
6      class Factura {
7          - id:integer{@id, @sequence}
8          - numero:integer
9      }
10
11     class TipoFactura {
12         - id:integer
13         - descripcion:string
14         /**
15          * Se utiliza para establecer cual sera el porcentaje
16          * del impuesto que se cobrara en la factura.
17          */
18         - pImpuesto:double
19     }
20
21     class Producto {
22         - id:integer{@id, @sequence}
23         - cantidad:integer
24     }
25
26     class DescripcionProducto {
27         - id:integer{@id, @sequence}
28         - descripcion:string
29         - codigo:string
30         - precio:double
31     }
  
```

```

32
33 relationships {
34     Factura "*" <— "*" Producto: productos
35     Factura "*" <— "1" TipoFactura: tipoFactura
36     Producto: productos "0..*" — "1" DescripcionProducto: descripcion
37 }
38 #+ENDDRT #token fin modelo

```

Con éste modelo, el lenguaje, generaría las cuatro clases que se definieron para el mismo, más una que sería el punto de inicio para el programa Java, éste último se toma como el nombre del proyecto.

Las clases que se generan se presentan a continuación:

Fragmento 3.4: Java - Ejemplo 2, Clase generada Facturacion.java

```

1 package Facturacion;
2 public class Facturacion {
3     public static void main (String[] args) {
4         // Implementacion del modulo
5     }
6 }

```

Fragmento 3.5: Java - Ejemplo 2, Clase generada TipoFactura.java

```

1 package Facturacion;
2 public class TipoFactura {
3     private int id;
4     private String descripcion;
5     /**
6      * Se utiliza para establecer cual sera el porcentaje
7      * del impuesto que se cobrara en la factura.
8      */
9     private double pImpuesto;
10
11     // Constructor por defecto
12     public TipoFactura(int id, String descripcion,
13         double pImpuesto){
14         this.id = id;
15         this.descripcion = descripcion;
16         this.pImpuesto = pImpuesto;
17     }
18
19     public int getId(){
20         return this.id;
21     }
22
23     public String getDescripcion(){
24         return this.descripcion;
25     }
26
27     public void setDescripcion(String descripcion){
28         this.descripcion = descripcion;
29     }
30
31     public double getPImpuesto(){
32         return this.pImpuesto;
33     }

```

```

34
35     public void setPImpuesto(double pi){
36         this.pImpuesto = pi;
37     }
38 }

```

Fragmento 3.6: Java - Ejemplo 2, Clase generada `Producto.java`

```

1 package Facturacion;
2 public class Producto{
3     private int id;
4     private int cantidad;
5     private DescripcionProducto descripcion;
6
7     // Constructor por defecto
8     public Producto(int id, int cantidad,
9         DescripcionProducto descripcion) {
10         this.id = id;
11         this.cantidad = cantidad;
12         this.descripcion = descripcion;
13     }
14
15     public int getId(){
16         return this.id;
17     }
18
19     public int getCantidad(){
20         return this.cantidad;
21     }
22
23     public void setCantidad(int cantidad){
24         this.cantidad = cantidad;
25     }
26
27     public DescripcionProducto getDescripcion() {
28         return this.descripcion;
29     }
30
31     public void setDescripcion(DescripcionProducto dp){
32         this.descripcion = dp;
33     }
34 }

```

Fragmento 3.7: Java - Ejemplo 2, Clase generada `DescripcionProducto.java`

```

1 package Facturacion;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 public class DescripcionProducto{
7     private int id;
8     private String descripcion;
9     private String codigo;
10    private Double precio;
11    private List<Producto> productos = new ArrayList<>();
12 }

```



```

13 // Constructor por defecto
14 public DescripcionProducto(int id, String descripcion,
15     String codigo, Double precio) {
16     this.id = id;
17     this.descripcion = descripcion;
18     this.codigo = codigo;
19     this.precio = precio;
20 }
21
22 public int getId() {
23     return this.id;
24 }
25
26 public String getDescripcion(){
27     return this.descripcion;
28 }
29
30 public void setDescripcion(String descripcion){
31     this.descripcion = descripcion;
32 }
33
34 public String getCodigo() {
35     return this.codigo;
36 }
37
38 public void seCodigo(String codigo){
39     this.codigo = codigo;
40 }
41
42 public Double getPrecio(){
43     return this.precio;
44 }
45
46 public void setPrecio(Double precio){
47     this.precio = precio;
48 }
49
50
51 public List<Producto> getProductos(){
52     return this.productos;
53 }
54
55 public void quitarProducto(Producto p){
56     if (this.productos.contains(p)) {
57         this.productos.remove(p);
58     } else {
59         System.err.print("Error: no se encuentra el Producto "+p
60             +" en la lista de Productos en esta descripcion");
61     }
62 }
63
64 public void agregarProducto(Producto p) {
65     this.productos.add(p);
66 }
67 }

```

Fragmento 3.8: Java - Ejemplo 2, Clase generada Factura.java

```

1 package Facturacion;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 public class Factura{
7     private int id;
8     private int numero;
9     private TipoFactura tipoFactura;
10    private List<Producto> productos;
11
12    // Constructor por defecto
13    public Factura(int id, int numero
14        TipoFactura tipoFactura) {
15        this.id = id;
16        this.numero = numero;
17        this.productos = new ArrayList<>();
18    }
19
20    public int getId(){
21        return this.id;
22    }
23
24    public int getNumero() {
25        return this.numero;
26    }
27
28    public void setNumero(int numero){
29        this.numero = numero;
30    }
31
32    public TipoFactura getTipoFactura(){
33        return this.tipoFactura;
34    }
35
36    public void setTipoFactura(TipoFactura tipoFactura){
37        this.tipoFactura = tipoFactura;
38    }
39    public List<Producto> getProductos(){
40        return this.productos;
41    }
42
43    public void quitarProducto(Producto producto){
44        if (this.productos.contains(producto)) {
45            this.productos.remove(producto);
46        } else {
47            System.err.print("Error: no se encuentra el Producto "+
48                producto+" en la lista de Productos de la factura");
49        }
50    }
51
52    public void agregarProducto(Producto producto) {
53        this.productos.add(producto);
54    }
55 }

```

3.3. Ejemplo 3

En esta sección se abordará un ejemplo más extenso que los vistos anteriormente, aquí, se presenta un escenario real, utilizado como un trabajo integrador para la cátedra Programación Orientada a Objetos I, en el año 2017, dicha consigna fue confeccionada por el Lic. Claudio O. Biale.

Consigna:

Una empresa de reparación de artículos, por ejemplo, hardware de computadoras desea implementar un sistema de gestión de reclamos.

El sistema debe soportar el ingreso de reclamos, la asignación de recursos para la reparación de los artículos y el seguimiento de las reparaciones.

El sistema deberá registrar para cada reclamo un número, la descripción del problema, el tipo de artículo a reparar, la fecha en la que fue ingresado al sistema y la fecha estimada de entrega. De los tipos de artículos interesa registrar un código (que lo identifica) y un nombre.

Los reclamos serán reparados por los técnicos de la empresa, los cuales deberán estar registrados en el sistema. De cada técnico interesa saber su documento único, nombres, apellidos y los tipos de artículos sobre los que está capacitado para trabajar. Un técnico puede trabajar como empleado mensual o jornalero. En caso de ser jornalero interesa conocer la tarifa por hora que se le paga, mientras que para un empleado mensual interesa conocer el sueldo mensual que percibe.

Para realizar una reparación, se planifica de entre el conjunto de tareas definidas en el sistema, una secuencia de tareas que son las que los técnicos deberán de realizar para concluir la reparación. De las tareas interesa su código único, nombre, descripción y tipos de artículos a los que aplica. Cada tarea de una reparación será efectuada por un técnico de la empresa capacitado en el tipo de artículo a reparar.

Es de interés para la empresa llevar un registro del tiempo invertido por el técnico en cada tarea realizada. Por ello, será necesario registrar en el sistema la cantidad de horas dedicadas y el día que se realizó la tarea. Si la misma es efectuada en más de un día, interesa saber cuantas horas le dedicó para cada fecha. Se debe indicar cuando se finaliza la tarea.

Aquí se presenta el diagrama de clases que modela un posible resultado para el problema en cuestión.

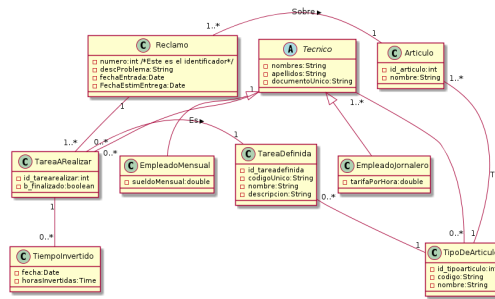


Figura 3.3: Diagrama de Clases - Modelo ejemplo 3

Y luego de pasarlo a un modelo Director se termina teniendo lo siguiente:

Fragmento 3.9: Director - Modelo equivalente a la Figura 3.3

```

1  #+DRT.CONFIG.O.LANG=java
2  #+DRT.CONFIG.O.PROJ=Reparaciones
3  #+JAVA.CONFIG.O.STRUCT=split
4
5  #+BEGINDRT
6  #{ Clase abstracta que permite que los diferentes tipos
7  de tecnicos compartan los atributos que tienen en comun}#
8  abstract class Tecnico {
9      - nombres:string
10     - apellidos:string
11     - documentoUnico:string
12 }
13
14 class EmpleadoJornalero {
15     - tarifaPorHora:double
16 }
17
18 class EmpleadoMensual {
19     - sueldoMensual:double
20 }
21
22 class TipoDeArticulo {
23     - id_tipoarticulo:integer
24     - codigo:string
25     - nombre:string
26 }
27
28 class Articulo {
29     - id_articulo:integer
30     - nombre:string
31 }
32
33 class TareaARealizar {
34     - id_tarearealizar:integer
35     - b_finalizado:boolean
36 }
37

```

```

38 class TareaDefinida {
39     - id_tareadefinida
40     - codigoUnico:string
41     - nombre:string
42     - descripcion:string
43 }
44
45 class Reclamo {
46     - numero:integer
47     - descProblema:string
48     - fechaEntrada:date
49     - FechaEstimEntrega:date
50 }
51
52 class TiempoInvertido {
53     - fecha:date
54     - horasInvertidas:Time
55 }
56
57 relationships {
58     Tecnico:tecnicos "1..*" — "*" TipoDeArticulo:artEsp
59     Tecnico:tecnico "1" — "*" TareaARealizar:tareas
60     Tecnico <|- EmpleadoJornalero
61     Tecnico <|- EmpleadoMensual
62     Reclamo:reclamos "*" — "1" Articulo:articulo
63     TareaARealizar:tareas "*" — "1" TareaDefinida:tarea
64     TareaDefinida:tDA "*" — "1" TipoDeArticulo:tAA
65     Reclamo:reclamo "1" — "*" TareaARealizar:tareas
66     TipoDeArticulo:tipo "1" — "*" Articulo:articulos
67     TareaARealizar:tarea "1" — "*" TiempoInvertido:tiempos
68 }
69
70 #+ENDDRT

```

A partir del modelo, Director pasará a generar las nueve clases descritas más una que será el punto de inicio (por el hecho de que la salida está configurada para ser Java).

Fragmento 3.10: Java - Ejemplo 3, Clase generada Reparaciones.java

```

1 package Reparaciones;
2 public class Reparaciones {
3     public static void main(String[] args) {
4         // Implementacion del modulo
5     }
6 }

```

Fragmento 3.11: Java - Ejemplo 3, Clase generada Reclamo.java

```

1 package Reparaciones;
2
3 import java.util.Date;
4 import java.util.List;
5 import java.util.ArrayList;
6
7 public class Reclamo{
8     private int numero;

```

```

9 private String descProblema;
10 private Date fechaEntrada;
11 private Date fechaEstimEntrega;
12 private Artículo articulo;
13 private List<TareaARealizar> tareas;
14
15 public Reclamo(int numero, String descProblema,
16     Date fechaEntrada, Date fechaEstimEntrega
17     Artículo articulo){
18     this.numero = numero;
19     this.descProblema = descProblema;
20     this.fechaEntrada = fechaEntrada;
21     this.fechaEstimEntrega = fechaEstimEntrega;
22     this.articulo = null;
23     this.tareas = new ArrayList<>();
24 }
25
26 public int getNumero() {
27     return this.numero;
28 }
29
30 public void setNumero(int numero) {
31     this.numero = numero;
32 }
33
34 public String getDescProblema() {
35     return this.descProblema;
36 }
37
38 public void setDescProblema(String descProblema) {
39     this.descProblema = descProblema;
40 }
41
42 public Date getFechaEntrada() {
43     return this.fechaEntrada;
44 }
45
46 public void setFechaEntrada(Date fechaEntrada) {
47     this.fechaEntrada = fechaEntrada;
48 }
49
50 public Date getFechaEstimEntrega() {
51     return this.fechaEstimEntrega;
52 }
53
54 public void setFechaEstimEntrega(Date fechaEstimEntrega) {
55     this.fechaEstimEntrega = fechaEstimEntrega;
56 }
57
58 public List<TareaARealizar> getTareas(){
59     return this.tareas;
60 }
61
62 public void quitarTarea(TareaARealizar tarea){
63     if (this.tareas.contains(tarea)) {
64         this.tareas.remove(tarea);
65     } else {

```

```

66         System.err.print("Error: no se encuentra la tarea "+
67             tarea+" en la lista de tareas del reclamo");
68     }
69 }
70
71 public void agregarTarea(TareaARealizar tarea) {
72     this.tareas.add(tarea);
73 }
74
75 public Articulo getArticulo() {
76     return this.articulo;
77 }
78
79 public void setArticulo(Articulo articulo) {
80     this.articulo = articulo;
81 }
82 }

```

Fragmento 3.12: Java - Ejemplo 3, Clase generada Articulo.java

```

1 package Reparaciones;
2
3 import java.util.List;
4 import java.util.ArrayList;
5
6 public class Articulo{
7     private String nombre;
8     private TipoDeArticulo tipo;
9     private List<Reclamo> reclamos;
10
11     public Articulo(String nombre){
12         this.nombre = nombre;
13         this.tipo = null;
14         this.reclamos = new ArrayList<>();
15     }
16
17     public String getNombre() {
18         return this.nombre;
19     }
20
21     public void setNombre(String nombre) {
22         this.nombre = nombre;
23     }
24
25     public Articulo getTipo() {
26         return this.tipo;
27     }
28
29     public void setTipo(TipoDeArticulo tipo ) {
30         this.tipo = tipo;
31     }
32
33     public List<Reclamo> getReclamos(){
34         return this.reclamos;
35     }
36
37     public void agregarReclamo(Reclamo reclamo) {

```

```

38     this.reclamos.add(reclamo);
39 }
40
41 public void quitarReclamo(Reclamo reclamo){
42     if (this.reclamos.contains(reclamo)) {
43         this.reclamos.remove(reclamo);
44     } else {
45         System.err.print("Error: no se encuentra el reclamo "+
46             reclamo+" en la lista de reclamos del articulo");
47     }
48 }
49 }

```

Fragmento 3.13: Java - Ejemplo 3, Clase generada TareaDefinida.java

```

1 package Reparaciones;
2
3 import java.util.List;
4 import java.util.ArrayList;
5
6 public class TareaDefinida{
7     private String codigoUnico;
8     private String nombre;
9     private String descripcion;
10    private TipoDeArticulo tAA;
11    private List<TareaARealizar> tareas = new ArrayList<>();
12
13    public TareaDefinida(String codigo, String nombre,
14        String descripcion) {
15        this.codigoUnico = codigo;
16        this.nombre = nombre;
17        this.descripcion = descripcion;
18        this.tAA = null;
19    }
20
21    public String getCodigoUnico() {
22        return this.codigoUnico;
23    }
24
25    public void setCodigoUnico(String codigoUnico) {
26        this.codigoUnico = codigoUnico;
27    }
28
29    public String getNombre() {
30        return this.nombre;
31    }
32
33    public void setNombre(String nombre) {
34        this.nombre = nombre;
35    }
36
37    public String getDescripcion() {
38        return this.descripcion;
39    }
40
41    public void setDescripcion(String descripcion) {
42        this.descripcion = descripcion;

```



```

43     }
44
45     public TipoDeArticulo getTipoArticuloAsociado() {
46         return this.tAA;
47     }
48
49     public void setTipoArticuloAsociado(TipoDeArticulo tAA) {
50         this.tAA = tAA;
51     }
52
53     public List<TareaARealizar> getTareas() {
54         return this.tareas;
55     }
56
57     public void agregarTarea(TareaARealizar tarea) {
58         this.tareas.add(tarea);
59     }
60
61     public void quitarTarea(TareaARealizar tarea){
62         if (this.tareas.contains(tarea)) {
63             this.tareas.remove(tarea);
64         } else {
65             System.err.print("Error: no se encuentra la tarea "+
66                 tarea+" en la lista de tareas de la tarea definida");
67         }
68     }
69 }

```

Fragmento 3.14: Java - Ejemplo 3, Clase generada TipoDeArticulo.java

```

1 package Reparaciones;
2
3 import java.util.List;
4 import java.util.ArrayList;
5
6 public class TipoDeArticulo{
7     private String codigo;
8     private String nombre;
9     private List<Articulo> articulos;
10    private List<Tecnico> tecnicos;
11    private List<TareaDefinida> tDA;
12
13    public TipoDeArticulo(String codigo, String nombre) {
14        this.codigo = codigo;
15        this.nombre = nombre;
16        this.articulos = new ArrayList<>();
17        this.tDA = new ArrayList<>();
18    }
19
20    public String getCodigo() {
21        return this.codigo;
22    }
23
24    public void setCodigo(String codigo) {
25        this.codigo = codigo;
26    }
27

```

```

28 public String getNombre() {
29     return this.nombre;
30 }
31
32 public void setNombre(String nombre) {
33     this.nombre = nombre;
34 }
35
36 public List<Articulo> getArticulos() {
37     return this.articulos;
38 }
39
40 public void agregarArticulo(Articulo articulo) {
41     this.articulos.add(articulo);
42 }
43
44 public void quitarArticulo(Articulo articulo) {
45     this.articulos.remove(articulo);
46 }
47
48 public List<TareaDefinida> getTDA() {
49     return this.tDA;
50 }
51
52 public void agregarTDA(TareaDefinida TDA) {
53     this.tDA.add(TDA);
54 }
55
56 public void quitarTDA(TareaDefinida TDA){
57     this.tDA.remove(TDA);
58 }
59
60 /*
61  * Aun no es posible manejar instancias de subclases que
62  * heredan de una clase abstracta, por lo tanto, es necesario
63  * que implemente de forma manual como tratar a la lista:
64  * List<Tecnico> tecnicos.
65  */
66 }

```

En el código anterior se ve que una de las relaciones que se tienen plasmadas en el **Fragmento 3.9** específicamente en la **línea 58** la cual representa la primer relación en el conjunto de relaciones que hacen al modelo, ésta, asocia a una **clase** con una **clase abstracta**, la cual, como se puede ver en las subsiguientes líneas 60 y 61 tiene subclases, lo cual genera mas complejidad a la hora de tratar la lista de tecnicos que se debe manejar en la clase descrita en el **Fragmento 3.14**, además, éste comportamiento no se describe a lo largo del documento, por lo que se opta, en la presente iteración del lenguaje hacer que se emita una notificación en el código generado, a modo de comentario, para que el usuario implemente lo necesario para tratar las instancias de las subclases de la clase abstracta.

Fragmento 3.15: Java - Ejemplo 3, Clase generada `TareaARealizar.java`

```

1 package Reparaciones;

```

```

2
3 import java.util.List;
4 import java.util.ArrayList;
5
6 public class TareaARealizar{
7     private boolean finalizado;
8     private Reclamo reclamo;
9     private List<TiempoInvertido> tiempos;
10    private TareaDefinida tarea;
11    private Tecnico tecnico;
12
13    public TareaARealizar(boolean finalizado, Reclamo reclamo,
14        TareaDefinida tarea, Tecnico tecnico){
15        this.finalizado = finalizado;
16        this.reclamo = reclamo;
17        this.tiempos= new ArrayList<>();
18        this.tarea = tarea;
19        this.tecnico = tecnico;
20    }
21
22    public boolean isFinalizado() {
23        return finalizado;
24    }
25
26    public void setFinalizado(boolean finalizado) {
27        this.finalizado = finalizado;
28    }
29
30    public Reclamo getReclamo() {
31        return this.reclamo;
32    }
33
34    public void setReclamo(Reclamo reclamo){
35        this.reclamo = reclamo;
36    }
37
38    public TareaDefinida getTarea() {
39        return this.tarea;
40    }
41
42    public void setTarea(TareaDefinida tarea){
43        this.tarea = tarea;
44    }
45
46    public Tecnico getTecnico() {
47        return this.tecnico;
48    }
49
50    public void setTecnico(Tecnico tecnico){
51        this.tecnico = tecnico;
52    }
53
54    public List<TiempoInvertido> getTiempos() {
55        return this.tiempos;
56    }
57
58    public void agregarTiempo(TiempoInvertido tiempo){

```

```

59     this.tiempos.add(tiempo);
60 }
61
62 public void quitarTiempo(TiempoInvertido tiempo) {
63     this.tiempos.remove(tiempo);
64 }
65 }

```

Fragmento 3.16: Java - Ejemplo 3, Clase generada TiempoInvertido.java

```

1 package Reparaciones;
2
3 import java.util.Date;
4
5 public class TiempoInvertido{
6     private double horasInvertidas;
7     private Date fecha;
8     private TareaARealizar tarea;
9
10    public TiempoInvertido(TareaARealizar tarea ,
11        double horasInvertidas , Date fecha) {
12        this.horasInvertidas = horasInvertidas;
13        this.fecha = null;
14        this.tarea = tarea;
15    }
16
17    public double getHorasInvertidas() {
18        return this.horasInvertidas;
19    }
20
21    public void setHorasInvertidas(double horasInvertidas) {
22        this.horasInvertidas = horasInvertidas;
23    }
24
25    public Date getFecha() {
26        return this.fecha;
27    }
28
29    public void setFecha(Date nFecha) {
30        this.fecha = nFecha;
31    }
32
33    public TareaARealizar getTarea() {
34        return this.tarea;
35    }
36
37    public void setTarea(TareaARealizar tarea) {
38        this.tarea = tarea;
39    }
40 }

```

Fragmento 3.17: Java - Ejemplo 3, Clase generada Tecnico.java

```

1 package Reparaciones;
2
3 import java.util.List;
4 import java.util.ArrayList;

```

```

5
6 public abstract class Tecnico{
7     protected String nombres;
8     protected String apellidos;
9     protected String documentoUnico;
10    protected boolean activo;
11    protected List<TipoDeArticulo> artEsp;
12    protected List<TareaARealizar> tareas;
13
14    protected Tecnico(String nombres, String apellidos ,
15        String documentoUnico, boolean activo) {
16        this.nombres = nombres;
17        this.apellidos = apellidos;
18        this.documentoUnico = documentoUnico;
19        this.activo = activo;
20        this.artEsp = new ArrayList<>();
21        this.tareas = new ArrayList<>();
22    }
23
24    protected abstract List<TipoDeArticulo> getArtEsp();
25    protected abstract void agregarArtEsp(TipoDeArticulo artEsp);
26    protected abstract void quitarArtEsp(TipoDeArticulo artEsp);
27
28    protected abstract List<TareaARealizar> getTareas();
29    protected abstract void agregarTarea(TareaARealizar tarea);
30    protected abstract void quitarTarea(TareaARealizar tarea);
31 }

```

Fragmento 3.18: Java - Ejemplo 3, Clase generada `EmpleadoMensual.java`

```

1 package Reparaciones;
2
3 import java.util.List;
4 import java.util.ArrayList;
5
6 public class EmpleadoMensual extends Tecnico{
7     private double sueldoMensual;
8
9     public EmpleadoMensual(String nombres, String apellidos ,
10        String documentoUnico, double sueldoMensual, boolean activo) {
11        super(nombres, apellidos, documentoUnico, activo);
12        this.sueldoMensual = sueldoMensual;
13    }
14
15    public String getNombre() {
16        return this.nombres;
17    }
18
19    public void setNombre(String nombres) {
20        this.nombres = nombres;
21    }
22
23    public String getApellidos() {
24        return this.apellidos;
25    }
26
27    public void setApellidos(String apellidos) {

```

```

28     this.apellidos = apellidos;
29 }
30
31 public String getDocumentoUnico() {
32     return this.documentoUnico;
33 }
34
35 public void setDocumentoUnico (String documentoUnico) {
36     this.documentoUnico = documentoUnico;
37 }
38
39 public void setSueldoMensual(double sueldoMensual) {
40     this.sueldoMensual = sueldoMensual;
41 }
42
43 public double getSueldoMensual() {
44     return this.sueldoMensual;
45 }
46
47 public boolean getActivo() {
48     return this.activo;
49 }
50
51 public void setActivo(boolean activo) {
52     this.activo = activo;
53 }
54
55 /* Implementacion metodos abstractos de la superclase*/
56 @Override
57 public List<TipoDeArticulo> getArtEsp() {
58     return this.artEsp;
59 }
60
61 @Override
62 public void agregarArtEsp(TipoDeArticulo artEsp){
63     this.articulosEspecializados.add(artEsp);
64 }
65
66 @Override
67 public void quitarArtEsp(TipoDeArticulo artEsp) {
68     this.articulosEspecializados.remove(artEsp);
69 }
70
71 @Override
72 public List<TareaARealizar> getTareas() {
73     return this.tareas;
74 }
75
76 @Override
77 public void agregarTarea(TareaARealizar tarea) {
78     this.tareas.add(tarea);
79 }
80
81 @Override
82 public void quitarTarea(TareaARealizar tarea) {
83     this.tareas.remove(tarea);
84 }

```

Fragmento 3.19: Java - Ejemplo 3, Clase generada EmpleadoJornalero.java

```
1 package Reparaciones;
2
3 import java.util.List;
4 import java.util.ArrayList;
5
6 public class EmpleadoJornalero extends Tecnico{
7     private double tarifaPorHora;
8
9     public EmpleadoJornalero(String nombres, String apellidos,
10         String documentoUnico, double tarifaPorHora, boolean activo) {
11         super(nombres, apellidos, documentoUnico, activo);
12         this.tarifaPorHora = tarifaPorHora;
13     }
14
15     public String getNombre() {
16         return this.nombres;
17     }
18
19     public void setNombre(String nombres) {
20         this.nombres = nombres;
21     }
22
23     public String getApellidos() {
24         return this.apellidos;
25     }
26
27     public void setApellidos(String apellidos) {
28         this.apellidos = apellidos;
29     }
30
31     public String getDocumentoUnico() {
32         return this.documentoUnico;
33     }
34
35     public void setDocumentoUnico (String documentoUnico) {
36         this.documentoUnico = documentoUnico;
37     }
38
39     public void setTarifaPorHora(double tarifaPorHora) {
40         this.tarifaPorHora = tarifaPorHora;
41     }
42
43     public double getTarifaPorHora() {
44         return this.tarifaPorHora;
45     }
46
47     public boolean getActivo() {
48         return this.activo;
49     }
50
51     public void setActivo(boolean activo) {
52         this.activo = activo;
53     }
```

```

54
55  /*Implementacion metodos abstractos de la superclase*/
56  @Override
57  public List<TipoDeArticulo> getArtEsp() {
58      return this.artEsp;
59  }
60
61  @Override
62  public void agregarArtEsp(TipoDeArticulo artEsp){
63      this.articulosEspecializados.add(artEsp);
64  }
65
66  @Override
67  public void quitarArtEsp(TipoDeArticulo artEsp) {
68      this.articulosEspecializados.remove(artEsp);
69  }
70
71  @Override
72  public List<TareaARealizar> getTareas() {
73      return this.tareas;
74  }
75
76  @Override
77  public void agregarTarea(TareaARealizar tarea) {
78      this.tareas.add(tarea);
79  }
80
81  @Override
82  public void quitarTarea(TareaARealizar tarea) {
83      this.tareas.remove(tarea);
84  }
85  }

```


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. Aldaej 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. Gullesten 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 Adaptive Object Models*, 2001.
- [8] S. Sobernig, B. Hoisl y M. Strembeck, «Extracting Reusable Design Decisions 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/>.
- [15] C. Larman, *Applying UML and Patterns, An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Addison Wesley Professional, 2004.
- [16] A. Dennis, B. H. Wixom y D. Tegarden, *System Analysis Design UML Version 2.0, An Object-Oriented Approach*. John Wiley y Sons, Inc., 2012.
- [17] B. Dathan y S. Ramnath, *Object-Oriented Analysis, Design and Implementation, An Integrated Approach*. USA: Springer, 2015.
- [18] J. W. Satzinger, R. B. Jackson y S. D. Burd, *System Analysis and Design, In a Changing World*. USA: Course Technology, Cengage Learning, 2012.
- [19] C. Larman, *UML y Patrones, Una introducción al análisis y diseño orientado a objetos y al proceso unificado*. Madrid: Pearson Educación, 2003.
- [20] M. Seidl, M. Scholz, C. Huemer y G. Kappel, *UML @ Classroom, An Introduction to Object-Oriented Modeling*. Heidelberg, Germany: Springer, 2012.
- [21] C. O. Biale, «Programación Orientada a Objetos II, Clase 10 - Diseño», en, 2017.