Desarrollo de un Generador de Código Swift a Partir de Modelos de Clase en UML 2

Tomás Balderas Contreras
balderas@ccc.inaoep.mx

5 de noviembre de 2018

NOTA DE DESCARGO DE RESPONSABILIDAD. El autor del presente documento no se hace responsable de daños a personas o bienes que ocurran por el uso malintencionado que terceros hagan de la información contenida en este reporte.

Tabla de Contenido

Desarrollo de un Generador de Código Swift a Partir de Modelos de Clase en UML 2

Tomás Balderas Contreras
balderas@ccc.inaoep.mx

Resumen

Este documento presenta al lector un tutorial básico para el uso de una tecnología que permite transformar modelos construidos a partir del lenguaje UML 2 en cualquier tipo de información textual, incluyendo código fuente en algún lenguaje de programación. La tecnología utilizada se llama Acceleo, funciona dentro del entorno de desarrollo Eclipse, y se complementa con otras tecnologías, como la plataforma Uml2 y la herramienta de modelado Papyrus. Como caso de estudio se presenta el desarrollo de un generador de código en lenguaje Swift a partir de diagramas de clase en UML 2.

1. Arquitectura dirigida por modelos

La construcción de modelos es una tarea crucial en diferentes áreas del quehacer humano, pues permiten representar sistemas complejos antes de implementarlos, así como estudiar fenómenos a distintos niveles de abstracción. Existen herramientas de software que apoyan en la elaboración y manipulación de modelos; por ejemplo, la herramienta Simulink permite construir modelos de sistemas utilizando componentes parametrizables básicos, así como ejecutar el modelo para analizar sus resultados.

El concepto de *ingeniería basada en modelos*, que ha cobrado relevancia en los últimos años, comprende el diseño e implementación de sistemas de software a partir de descripciones funcionales iniciales basadas completamente en modelos comprensibles tanto por los desarrolladores como por los stakeholders. Los objetivos de este paradigma de desarrollo son: paliar la complejidad de los sistemas actuales, e incrementar la productividad durante el diseño e implementación de software.

El Object Management Group (OMG) propuso una infraestructura de desarrollo de software basada en modelos, conocida como *arquitectura dirigida por modelos*, que consta de los siguientes elementos [4]:

- Lenguaje de modelado *independiente de la plataforma de implementación*: se construye a base de extender los elementos de modelado de UML 2 llano para representar conceptos y abstracciones de un *dominio de aplicación* específico de manera precisa. A este tipo de dialecto de UML 2 se le denomina *perfil*.
- Lenguaje de modelado *dependiente de la plataforma de implementación*: se construye a base de extender los elementos de UML 2 llano para representar componentes y elementos propios de un framework de desarrollo de software específico.
- Herramienta de *transformación modelo a modelo*: es software que procesa un modelo expresado en lenguaje de modelado independiente de la plataforma y lo transforma en el modelo expresado en lenguaje dependiente de la plataforma que lo implementa.

- Herramienta de *transformación modelo a texto*: es software que genera código fuente en un lenguaje de programación específico a partir del modelo dependiente de la plataforma de implementación.

La Figura 1 muestra la aplicación de diferentes arquitecturas dirigidas por modelos para implementar un sistema de software utilizando diferentes frameworks y lenguajes de implementación. El objetivo es incrementar la productividad mediante el diseño de un único modelo independiente de la plataforma de implementación y el uso de diferentes herramientas de transformación.

Actualmente están disponibles un conjunto de tecnologías que operan en el entorno *Eclipse*, y que permiten construir arquitecturas dirigidas por modelos como las mostradas en la Figura 1. La mancuerna *Uml2* y *Papyrus* [7] permite construir diagramas acordes a las reglas de UML 2, almacenar modelos en un formato adecuado para su procesamiento por herramientas de transformación (o simplemente transformaciones), y definir y aplicar perfiles de UML 2; la tecnología *ATL* (ATL Transformation Language) permite desarrollar transformaciones modelo a modelo; y la tecnología *Acceleo* permite escribir transformaciones modelo a texto para la generación de código. Estas tecnologías están respaldadas por estándares definidos y publicados por el OMG.

El contenido restante de este documento describe cómo utilizar la tecnología Acceleo para desarrollar una sencilla transformación que genera código en lenguaje Swift [1] a partir de modelos de clases en UML 2 construidos con Uml2 y Papyrus. El tutorial está destinado a ingenieros de software con conocimientos sólidos en teoría de orientación a objetos, familiarizados con el uso de lenguajes para desarrollo orientados a objetos, y con experiencia en modelado con UML 2.

2. Modelado y metamodelado en UML 2

Utilizaremos el modelo de clases mostrado en la Figura 2 para ilustrar conceptos y probar la transformación descrita en la siguiente sección. El diagrama categoriza los elementos que participan en una aerolínea: las personas se clasifican en pasajeros y empleados, los pasajeros se categorizan de acuerdo con su edad, y los empleados de acuerdo con sus funciones y si forman parte de la tripulación de una aeronave; los vehículos en la flota de la aerolínea se clasifican en aeronaves de pasajeros, de carga y de carga/pasajeros, y cada una de estas clases agrupa a unidades de diferentes fabricantes; finalmente, el modelo define clases que describen los vuelos gestionados y los aeropuertos registrados, junto con otras clases y enumeraciones de apoyo.

2.1. Estructura jerárquica de un modelo de clases en UML 2.

El diagrama de la Figura 2 es un ensamble jerárquico de varios elementos de modelado: cajas que representan enumeraciones y contienen elementos de modelado que representan literales; cajas que representan clases y contienen elementos de modelado que representan propiedades y otros que describen operaciones; líneas que representan relaciones (generalización y asociación) entre clases; etc. Todos estos elementos de modelado son objetos que el diseñador selecciona en una paleta proporcionada por Papyrus, coloca en el área de trabajo, configura de acuerdo con los requerimientos, y vincula hasta construir un diagrama completo.

Por ejemplo, el elemento de modelado utilizado para definir la clase `Persona` es un objeto que tiene como atributos el nombre de la clase que representa, una colección de elementos de modelado que representan las propiedades de una persona (nombre, fecha de nacimiento, sexo, número de pasaporte, etc.), una colección de elementos de modelado que representan las operaciones de la clase (las operaciones `init()`), entre muchos otros.

2.2. El metamodelo de UML 2

Un modelo describe elementos o fenómenos del mundo real, mientras que un *metamodelo* describe los elementos constitutivos del modelo: el metamodelo modela al modelo. El metamodelo de UML 2, documentado completamente en la especificación expedida por el OMG [6], contiene un extenso conjunto de *metaclases* que describen los elementos de modelado que componen todo diagrama acorde a UML 2: los elementos de modelado son instancias de las metaclases.

El metamodelo de UML 2 contiene una jerarquía de metaclases sumamente compleja, en la que cada metaclase hereda rasgos característicos de sus súper-metaclases y define rasgos propios. Por ejemplo, la Figura 3 muestra, de forma muy simplificada, la jerarquía para `Class`, la metaclase concreta que describe los elementos de modelado utilizados para definir las clases `Persona`, `Aeronave`, `Vuelo`, etc. en el diagrama de la Figura 2. Gracias a los rasgos que `Class` hereda, sus instancias son capaces de funcionar como elementos de modelado, tener un nombre, definir un espacio de nombres, describir un nuevo tipo de datos, poder ser redefinidas por una subclase, etc.

Las instancias de la metaclase `Class` también tienen la capacidad de contener un conjunto de propiedades, elementos de modelado instancias de la metaclase concreta `Property`, y un conjunto de operaciones, elementos de modelado instancias de la metaclase concreta `Operation`. La Figura 4 muestra, de forma simplificada, parte de la configuración de la instancia de `Class` utilizada para definir la clase `Persona`, y la relación de composición entre dicho objeto y las instancias de `Property` y `Operation`.

Uml2 es una implementación del metamodelo de UML 2 utilizada por la herramienta de modelado Papyrus para construir diagramas de forma visual y mediante manipulación directa. Papyrus y Uml2 no solo presentan una vista gráfica del modelo, también construyen una representación del diagrama en formato XML que puede ser procesada por las transformaciones. Lo anterior distingue a las herramientas de modelado de los programas para trazado de diagramas como Microsoft Visio, OmniGraffle y Dia.

2.3. Justificación del estudio del metamodelo

¿Por qué es necesario conocer el metamodelo de UML 2 para implementar generadores de código? Porque el proceso de transformar un modelo en texto consiste en tomar cada uno de los objetos constitutivos de un diagrama como el de la Figura 2, acceder a los valores de sus atributos e invocar la ejecución de sus operaciones para obtener información de su estructura y sus vínculos con otros objetos, lo que permite tomar decisiones y obtener datos que se integran al código producido. Para llevar a cabo lo anterior debemos conocer los elementos de modelado estudiando sus metaclases.

3. Generación de código utilizando Acceleo.

Acceleo [3] es una implementación de código abierto del estándar *MOF Model to Text Transformation* (MOFM2T) [5], expedido por el OMG. Los siguientes apartados describen cómo utilizar esta tecnología para generar código en Swift a partir de los elementos de modelado que constituyen los diagramas de clases. La transformación presta atención a las configuraciones de estos elementos de modelado, establecidas por el diseñador, y genera el código fuente correspondiente.

3.1. Definición del módulo principal de la transformación

El *módulo* es la construcción principal de nuestra transformación; es el contenedor y espacio de nombres para las construcciones que generan secciones de código (plantillas), y las construcciones que proporcionan información sobre el modelo fuente (consultas); puede importar otros módulos para utilizar sus construcciones y extender otro módulo para redefinir sus construcciones. En el siguiente segmento de código se utiliza la etiqueta `[module]` para declarar que el nombre del módulo para nuestra transformación es `generator()`, y que sus construcciones operan sobre modelos construidos a partir de Uml2, la implementación del metamodelo de UML 2 mencionada anteriormente.

```
1.  [comment encoding = UTF-8 /]
2.  [**
3.   * SwiftGen
4.   *
5.   * Generador de código UML 2 - Swift 4
6.   * Tomás Balderas Contreras
7.   * 2018
8.   *
9.   */]
10. [module generator('http://www.eclipse.org/uml2/5.0.0/UML')]
```

Todas las construcciones que componen el módulo se almacenan en el archivo `generator.mtl`, utilizado por Acceleo para ejecutar la transformación y generar código Swift a partir de un diagrama de clases fuente construido con Papyrus. Estas construcciones se agrupan en dos categorías: primero, las construcciones que producen código en Swift para definir enumeraciones; segundo, las construcciones que producen código en Swift para definir clases.

3.2. Definición de enumeraciones

Este apartado describe las construcciones en el módulo `generator()` que producen archivos de código en Swift con la definición de enumeraciones, a las que denominamos *enumeraciones objetivo*, a partir de los elementos equivalentes en el modelo fuente, a los que denominamos *enumeraciones fuente*.

3.2.1. Descripción de la transformación

El código de las construcciones de interés se presenta a continuación; para el modelo de la Figura 2, estas construcciones se aplican a cada una de las enumeraciones `EnumBanco`, `EnumIdioma`, `EnumPais`, `EnumCiudad`, `EnumMes`, etc. y crean el archivo con extensión ".swift" correspondiente.

```
1.  [**
2.   * Calcula el nombre de una instancia de Classifier (clases y
3.   * enumeraciones, entre otras) en formato UpperCamelCase.
4.   *
5.   * @param aClassifier
6.   */]
7.  [query private name(aClassifier: Classifier): String =
8.  aClassifier.name.toUpperFirst()
9.  /]
10.
11. [**
12.  * Plantilla principal - Genera un archivo en lenguaje Swift con
13.  * la declaración de una enumeración.
14.  *
15.  * @param anEnumeration
16.  */]
17. [template public generateEnumDeclaration(anEnumeration : Enumeration)]
18. [comment @main/]
19. [file (anEnumeration.name().concat('.swift'), false, 'UTF-8')]
20.   enum [anEnumeration.name()/] {
21.   [for (aLiteral: EnumerationLiteral | anEnumeration.ownedLiteral)]
22.     [aLiteral.generateEnumLiteral()/]
23.   [/for]
24.   }
25. [/file]
26. [/template]
27.
28. [**
29.  * Genera una línea de código con la declaración de una literal
30.  * de una enumeración.
31.  *
32.  * @param aLiteral
33.  */]
34. [template private generateEnumLiteral(aLiteral: EnumerationLiteral)
35.             post(trim())]
```

```
36.     case [aLiteral.name/]
37.     [/template]
```

Una *consulta* es una construcción que recibe elementos de modelado como parámetros y regresa un valor calculado en base a ellos; se define al interior de la etiqueta `[query …/]`. Una *plantilla* es una construcción que genera texto utilizando los elementos de modelado que recibe como parámetros; su definición se delimita por las etiquetas `[template …][/template]`. Algunos de los caracteres que componen el texto que produce la plantilla se indican explícitamente en su cuerpo, el resto resultan de aplicar otras plantillas o consultas sobre los parámetros.

La consulta `name()`, definida en las líneas 7–9, opera sobre una instancia de la metaclase `Classifier` y regresa un valor de tipo primitivo `String`, que se calcula cambiando la primera letra del nombre de la instancia a mayúscula y el resto a minúsculas. `Classifier` es una metaclase abstracta, así que la consulta se aplica a instancias de metaclases concretas que son subclases de `Classifier`, como `Class` y `Enumeration`. Por lo tanto, la consulta `name()` se puede utilizar indistintamente sobre elementos de modelado que definen clases o enumeraciones.

La plantilla `generateEnumDeclaration()`, definida en las líneas 17–26, se invoca por cada enumeración fuente, instancia de la metaclase `Enumeration`, en el diagrama. La plantilla crea un archivo de código con extensión ".swift" y coloca ahí el código para la enumeración objetivo. El código producido es una combinación de los caracteres explícitos contenidos en el cuerpo de la plantilla, el texto generado al aplicar la consulta `name()` sobre el parámetro, y el texto producido por la ejecución iterativa de la plantilla `generateEnumLiteral()`.

Cada instancia de `Enumeration` cuenta con el atributo `ownedLiteral`: una colección, posiblemente vacía, de instancias de la metaclase `EnumerationLiteral` que almacena las literales de la enumeración fuente especificadas en el modelo. Para generar la declaración de cada literal, la plantilla `generateEnumDeclaration()` itera sobre `ownedLiteral` utilizando la estructura de control `[for …][/for]`.

La plantilla `generateEnumLiteral()`, definida en las líneas 34–37, produce una línea de código que declara una literal de una enumeración fuente. La plantilla recibe como parámetro una instancia de `EnumerationLiteral`, y utiliza el valor del atributo `name` para generar el identificador de la literal; no puede utilizar la consulta `name()` porque `EnumerationLiteral` no es subclase de `Classifier`.

El lenguaje Swift permite especificar valores en bruto (raw values) y valores asociados (associated values) para las literales de enumeraciones; construcciones que no tienen equivalente en la definición de la metaclase `EnumerationLiteral`. Para subsanar esta limitación es necesario construir un perfil que extienda `EnumerationLiteral`, entre otros elementos de modelado, con propiedades y restricciones que permitan modelar, de forma precisa, construcciones del lenguaje Swift que no están presentes en UML 2 llano.

3.2.2. Análisis de resultados

A continuación, se muestra el código en lenguaje Swift producido por las plantillas `generateEnumDeclaration()` y `generateEnumLiteral()`, descritas anteriormente, para los elementos de modelado que definen las enumeraciones `EnumMes` y `EnumIdioma`. Se observa en el listado de código que las plantillas leen correctamente los valores de los atributos de los objetos en el diagrama, como su nombre y su colección de literales, y los incluye en el código producido.

```
1.  enum EnumMes {
2.      case enero
3.      case febrero
4.      case marzo
5.      case abril
6.      case mayo
7.      case junio
8.      case julio
9.      case agosto
10.     case septiembre
11.     case octubre
12.     case noviembre
13.     case diciembre
14.     case noEspecificado
15. }
16.
17. enum EnumIdioma {
18.     case ingles
19.     case espanol
20.     case aleman
21.     case frances
22.     case portugues
23.     case japones
24.     case chino
25.     case sueco
26.     case ruso
27.     case noEspecificado
28. }
```

3.3. Definición de clases

Este apartado describe las construcciones en el módulo `generator()` que producen archivos de código en Swift con la definición de clases, denominadas *clases objetivo*, a partir de los elementos equivalentes en el modelo fuente, denominados *clases fuente*.

3.3.1. Descripción de la transformación

El código de las construcciones de interés se presenta a continuación; para el modelo de la Figura 2, estas construcciones se aplican a cada una de las clases `Persona`, `Aeronave`, `Vuelo`, `Aeropuerto`, `Fecha`, etc. y crean el archivo con extensión ".swift" correspondiente.

```
1.  [**
2.   * Plantilla principal - Genera un archivo en lenguaje Swift con
3.   * la declaración de una clase sin súper-clase.
4.   *
5.   * @param aClass
6.   */]
7.  [template public generateClassDeclaration(aClass : Class) ?
8.                                  (aClass.superClass->isEmpty())]
9.  [comment @main/]
10. [file (aClass.name().concat('.swift'), false, 'UTF-8')]
11.   class [aClass.name()/] {
12.   [for (aProperty: Property | aClass.ownedAttribute)]
13.     [aProperty.generatePropertyDeclaration()/]
14.   [/for]
15.
16.   [for (anOperation: Operation | aClass.ownedOperation)]
17.     [anOperation.generateOperationDeclaration()/]
18.   [/for]
19.   }
20. [/file]
21. [/template]
22.
23. [**
24.   * Plantilla principal - Genera un archivo en lenguaje Swift con
25.   * la declaración de una clase con una única súper-clase.
26.   *
27.   * @param aClass
28.   */]
29. [template public generateClassDeclaration(aClass : Class) ?
30.                             (aClass.superClass->size() = 1)
31.                             {superClass: Class = aClass.superClass->any(true);}]
32. [comment @main/]
33. [file (aClass.name().concat('.swift'), false, 'UTF-8')]
34.   class [aClass.name()/]: [superClass.name()/] {
35.   [for (aProperty: Property | aClass.ownedAttribute)]
```

```
36.         [aProperty.generatePropertyDeclaration()/]
37.       [/for]
38.
39.       [for (anOperation: Operation | aClass.ownedOperation)]
40.         [anOperation.generateOperationDeclaration()/]
41.       [/for]
42.       }
43.     [/file]
44.     [/template]
```

Las plantillas que comparten el nombre `generateClassDeclaration()` reciben instancias de la metaclase `Class` como parámetro, y crean un archivo de código con extensión ".swift" donde colocan la definición de la clase objetivo correspondiente. La plantilla definida en las líneas 7–21 transforma clases fuente sin superclase; por ejemplo, `Persona`, `Vuelo` y `Aeropuerto` en el diagrama de la Figura 2. La plantilla definida en las líneas 29–44 transforma clases fuente con una única superclase; por ejemplo, `Empleado`, `Pasajero` y `Tripulante`.

La *condición de guarda*, o *precondición*, de una plantilla se sigue a la lista de parámetros y restringe la ejecución al cumplimiento de requerimientos específicos por parte de los parámetros. En el caso de la plantilla en las líneas 7–21, el requerimiento es que el atributo `superClass`, una colección de instancias de `Class` en el modelo fuente que son superclases del parámetro, no tenga elementos. Para la plantilla en las líneas 29–44, el requerimiento es que la propiedad `superClass` tenga un único elemento.

El texto producido por las plantillas es idéntico, salvo por la implementación de la relación de generalización entre la clase y la superclase, mediante el mecanismo de herencia de Swift, en la segunda plantilla. Ambas plantillas iteran sobre los atributos `ownedAttribute`, colección de instancias de la metaclase `Property`, y `ownedOperation`, colección de instancias de la metaclase `Operation`, de la clase fuente recibida como parámetro, y por cada elemento de modelado invocan la plantilla que produce el código con la declaración correcta.

Declaración de propiedades

A las propiedades de clases en Swift las denominamos *propiedades objetivo* a partir de este momento; a los elementos de modelado equivalentes en los modelos fuente los denominamos *propiedades fuente*, y son instancias de la metaclase `Property`. Una propiedad objetivo tiene variantes: puede ser constante o variable; de ser variable, puede ser almacenada o calculada; su valor puede ser una instancia o una colección de instancias; se le puede asignar un valor inicial o no; y se puede declarar como opcional o no. A continuación, se describe cómo las propiedades fuente modelan estas variantes, y cómo se produce la declaración de las propiedades objetivo.

La plantilla `generatePropertyDeclaration()`, listada a continuación, produce código a partir de instancias de `Property` que cumplen con el requisito de tener un tipo de datos especificado. La plantilla utiliza el valor del atributo `isReadOnly` de las propiedades fuente,

junto con la estructura de control `[if][else][/if]`, para decidir entre producir la declaración de una constante o de una variable. El diseñador establece el valor de `isReadOnly` para cada propiedad durante la construcción del modelo; si es verdadero, la propiedad no se puede modificar después de haberle asignado un valor inicial, que corresponde a la semántica de las constantes en Swift.

```
1.  [**
2.   * Genera una línea de código con la declaración de una constante
3.   * o una variable, dependiendo si la propiedad es de solo lectura
4.   * o no; todo lo anterior siempre y cuando la propiedad tenga un
5.   * tipo definido.
6.   *
7.   * @param aProperty
8.   */]
9.  [template private generatePropertyDeclaration(aProperty: Property) ?
10.                                 (aProperty.type <> null)
11.                                 post(trim())]
12.  [if (aProperty.isReadOnly)]
13.  [aProperty.generateConstantDeclaration()/]
14.  [else]
15.  [aProperty.generateVariableDeclaration()/]
16.  [/if]
17.  [/template]
```

La *multiplicidad* es un rasgo que indica el número de valores que puede almacenar una propiedad fuente; es un intervalo delimitado por los atributos `lower` y `upper`, heredados por `Property` de la metaclase abstracta `MultiplicityElement`. A continuación, se muestra el código de un par de consultas que proporcionan información relevante sobre la multiplicidad de las propiedades fuente. La consulta `isSingleValued()` verifica si la propiedad almacena un único valor. La consulta `isOptional()` verifica si la propiedad almacena un valor o ninguno, lo que permite modelar el concepto de *opcionales* en UML 2.

```
1.  [**
2.   * Verifica la multiplicidad de un elemento y determina si hace referencia
3.   * a un solo valor.
4.   *
5.   * @param anElement
6.   */]
7.  [query private isSinglevalued(anElement: MultiplicityElement): Boolean =
8.  anElement.lower = 1 and anElement.upper = 1
9.  /]
10.
11.  [**
```

```
12.     * Verifica la multiplicidad de un elemento y determina si hace referencia
13.     * a un valor o a ninguno.
14.     *
15.     * @param anElement
16.     */]
17.     [query private isOptional(anElement: MultiplicityElement): Boolean =
18.     anElement.lower = 0 and anElement.upper = 1
19.     /]
```

La plantilla `name()` produce el nombre correcto del tipo de datos que recibe como parámetro, y viene en dos variantes, listadas a continuación. La variante definida en las líneas 8–19 genera el nombre de alguno de los tipos primitivos de Swift en base al nombre del tipo primitivo de UML 2 que recibe como parámetro. La variante definida en las líneas 27–30 genera el nombre de una clase o enumeración presente en el modelo fuente utilizando la consulta `name()`, apropiada para tales elementos de modelado. Esta plantilla se aplica sobre el atributo `type` de las propiedades fuente, atributo que `Property` hereda de la metaclase abstracta `TypedElement`, cuyo valor lo establece el diseñador durante la construcción del modelo.

```
1.      [**
2.       * Genera el nombre de un tipo de datos primitivo en Swift 4 en base al
3.       * nombre del tipo de datos primitivo en UML 2 sobre el que se aplica
4.       * la plantilla.
5.       *
6.       * @param aType
7.       */]
8.      [template private name(aType: Type) ? (aType.oclIsKindOf(PrimitiveType)) post(trim())
9.                      {typeName: String = aType.name;}]
10.     [if (typeName = 'String')]
11.     String
12.     [elseif (typeName = 'Integer')]
13.     Int
14.     [elseif (typeName = 'Real')]
15.     Double
16.     [elseif (typeName = 'Boolean')]
17.     Bool
18.     [/if]
19.     [/template]
20.
21.     [**
22.      * Genera el nombre de un tipo de datos que es instancia de Classifier
23.      * (clase o enumeración, entre otras) en formato UpperCamelCase.
24.      *
```

```
25.      * @param aType
26.      */]
27.     [template private name(aType: Type) ? (aType.oclIsKindOf(Classifier)) post(trim())
28.                         {aClassifier: Classifier = aType.oclAsType(Classifier);}]
29.     [aClassifier.name()/]
30.     [/template]
```

La plantilla `generateConstantDeclaration()` produce una línea de código con la declaración de una propiedad objetivo constante, y viene en dos variantes, listadas a continuación. La variante definida en las líneas 9–19 produce la declaración de la constante seguida de la asignación de un valor inicial, siempre y cuando la propiedad fuente tenga definido uno. La variante definida en las líneas 29–39 produce únicamente la declaración de la constante. En ambos casos, la plantillas utilizan la estructura de control `[if][elseif][elseif][/if]` para decidir entre generar la declaración de una constante, de una constante opcional o de una constante que almacena una colección de objetos.

```
1.      [**
2.       * Genera una línea de código con la declaración de una constante
3.       * y la asignación de un valor a ella. La plantilla decide entre
4.       * generar la definición de una constante almacenada, de una constante
5.       * almacenada de tipo opcional o de una colección de elementos.
6.       *
7.       * @param aProperty
8.       */]
9.      [template private generateConstantDeclaration(aProperty: Property) ?
10.                                 (aProperty.defaultValue <> null)
11.                                 post(trim())]
12.     [if (aProperty.isSinglevalued())]
13.     let [aProperty.name/]: [aProperty.type.name()/] = [aProperty.defaultValue.generate()/]
14.     [elseif (aProperty.isOptional())]
15.     let [aProperty.name/]: [aProperty.type.name()/]? = [aProperty.defaultValue.generate()/]
16.     [elseif (aProperty.isMultivalued())]
17.     [aProperty.generateConstantCollectionDeclaration()/] = ['['/]['']'/]
18.     [/if]
19.     [/template]
20.
21.     [**
22.      * Genera una línea de código con la declaración de una constante.
23.      * La plantilla decide entre generar la declaración de una constante
24.      * almacenada, de una constante almacenada de tipo opcional o de una
25.      * colección de elementos.
26.      *
```

```
27.      * @param aProperty
28.      */]
29.     [template private generateConstantDeclaration(aProperty: Property) ?
30.                                     (aProperty.defaultValue = null)
31.                                     post(trim())]
32.     [if (aProperty.isSinglevalued())]
33.     let [aProperty.name/]: [aProperty.type.name()/]
34.     [elseif (aProperty.isOptional())]
35.     let [aProperty.name/]: [aProperty.type.name()/]?
36.     [elseif (aProperty.isMultivalued())]
37.     [aProperty.generateConstantCollectionDeclaration()/]
38.     [/if]
39.     [/template]
```

Para producir la declaración de una propiedad objetivo que almacena una colección de objetos, es necesario determinar el tipo de colección en base a la configuración de la propiedad fuente, establecida por el diseñador. Los atributos `isOrdered` e `isUnique`, heredados por `Property` de la metaclase abstracta `MultiplicityElement`, indican si los elementos de la colección están ordenados secuencialmente y si no están repetidos, respectivamente. La consulta `isSet()`, listada a continuación, verifica si una colección es un conjunto de elementos no ordenados ni duplicados. La consulta `isSequence()`, listada a continuación, verifica si una colección es una secuencia de elementos enumerados que puede contener elementos repetidos.

```
1.      [**
2.       * Verifica si un elemento hace referencia a una colección no ordenada
3.       * de valores únicos, es decir, a un conjunto.
4.       *
5.       * @param anElement
6.       */]
7.      [query private isSet(anElement: MultiplicityElement): Boolean =
8.      anElement.isOrdered._not() and anElement.isUnique
9.      /]
10.
11.     [**
12.      * Verifica si un elemento hace referencia a una colección numerada de
13.      * valores que se pueden repetir, es decir, a una secuencia.
14.      *
15.      * @param anElement
16.      */]
17.     [query private isSequence(anElement: MultiplicityElement): Boolean =
18.     anElement.isOrdered and anElement.isUnique._not()
19.     /]
```

La plantilla `generateConstantCollectionDeclaration()` recibe como parámetro una instancia de `Property`, y produce una línea de código con la declaración de una colección constante de instancias del tipo indicado por la propiedad fuente. La colección declarada puede ser de uno de dos tipos existentes en Swift: el *conjunto*, una colección de elementos no ordenados que no contiene elementos repetidos, o el *arreglo*, una colección de elementos secuenciales enumerados que puede almacenar elementos repetidos.

```
1.  [**
2.   * Genera la declaración de una colección constante de elementos
3.   * del mismo tipo. La plantilla decide entre declarar un arreglo
4.   * de elementos o un conjunto de elementos.
5.   *
6.   * @param aProperty
7.   */]
8.  [template private generateConstantCollectionDeclaration(aProperty: Property)
9.                                        post(trim())]
10. [if (aProperty.isSequence())]
11. let [aProperty.name/]: Array<[aProperty.type.name()/]>
12. [elseif (aProperty.isSet())]
13. let [aProperty.name/]: Set<[aProperty.type.name()/]>
14. [/if]
15. [/template]
```

La plantilla `generateVariableDeclaration()` produce una línea de código con la declaración de una propiedad objetivo variable; y viene en dos variantes, listadas más adelante. La variante definida en las líneas 10–19 produce la declaración de la variable seguida de la asignación de un valor inicial, siempre y cuando la propiedad fuente tenga definido uno. La variante definida en las líneas 29–42 produce únicamente la declaración de la variable. En ambos casos, la plantillas deciden entre generar la declaración de una variable, de una variable opcional o de una variable que almacena una colección de objetos.

Una *propiedad derivada* en UML 2 no almacena explícitamente un valor, sino que lo calcula a partir de los valores de otras propiedades; se distingue en un diagrama de clases porque su identificador está precedido por el símbolo "/". Cuando el diseñador especifica una propiedad derivada, la herramienta de modelado asigna el valor verdadero al atributo `isDerived` de la propiedad fuente. El concepto equivalente en Swift es la *propiedad calculada*, cuyo valor se determina a partir de los valores de otras propiedades de la clase. La plantilla definida en las líneas 29–42 verifica el valor del atributo `isDerived` para decidir si tiene que producir la declaración de una propiedad calculada.

```
1.  [**
2.   * Genera una línea de código con la declaración de una variable y
3.   * la asignación de un valor por omisión a ella. La plantilla
```

```
4.      * decide entre generar la definición de una variable almacenada,
5.      * de una variable almacenada de tipo opcional o de una colección
6.      * de elementos.
7.      *
8.      * @param aProperty
9.      */]
10.     [template private generateVariableDeclaration(aProperty: Property) ?
11.                     (aProperty.defaultValue <> null) post(trim())]
12.     [if (aProperty.isSinglevalued())]
13.     var [aProperty.name/]: [aProperty.type.name()/] = [aProperty.defaultValue.generate()/]
14.     [elseif (aProperty.isOptional())]
15.     var [aProperty.name/]: [aProperty.type.name()/]? = [aProperty.defaultValue.generate()/]
16.     [elseif (aProperty.isMultivalued())]
17.     [aProperty.generateVariableCollectionDeclaration()/] = ['['/]][']'/]
18.     [/if]
19.     [/template]
20.
21.     [**
22.      * Genera una línea de código con la declaración de una variable.
23.      * La plantilla decide entre generar la declaración de una variable
24.      * almacenada, de una variable calculada, de una variable almacenada
25.      * de tipo opcional o de una colección de elementos.
26.      *
27.      * @param aProperty
28.      */]
29.     [template private generateVariableDeclaration(aProperty: Property) ?
30.                     (aProperty.defaultValue = null) post(trim())]
31.     [if (aProperty.isSinglevalued())]
32.         [if (aProperty.isDerived)]
33.     [aProperty.generateComputedPropertyDeclaration()/]
34.         [else]
35.     var [aProperty.name/]: [aProperty.type.name()/]
36.         [/if]
37.     [elseif (aProperty.isOptional())]
38.     var [aProperty.name/]: [aProperty.type.name()/]?
39.     [elseif (aProperty.isMultivalued())]
40.     [aProperty.generateVariableCollectionDeclaration()/]
41.     [/if]
42.     [/template]
```

La plantilla `generateComputedPropertyDeclaration()`, listada a continuación, produce la declaración de una propiedad objetivo calculada. La plantilla genera el nombre y el tipo

de la propiedad objetivo, y las secciones `get{}` y `set{}` de la declaración. La única sentencia producida para la sección `get{}` regresa una instancia del mismo tipo que el de la propiedad objetivo. En teoría, se puede especificar el código para ambas secciones utilizando comentarios en el diagrama de clases fuente, que la plantilla puede incluir en la declaración de la propiedad.

```
1.  [**
2.   * Genera la declaración de una propiedad calculada. Para la sección
3.   * get, la plantilla genera una sentencia que regresa una instancia del
4.   * tipo de la propiedad calculada.
5.   *
6.   * @param aProperty
7.   */]
8.  [template private generateComputedPropertyDeclaration(aProperty: Property) post(trim())]
9.  var [aProperty.name/]: [aProperty.type.name()/] {
10.     get { return [aProperty.type.name()/]() }
11.     set {}
12. }
13. [/template]
```

Si una propiedad fuente tiene especificado un valor por omisión, la transformación lo considera como el valor inicial de la correspondiente propiedad objetivo. Cuando el diseñador especifica un valor por omisión para una propiedad fuente, la herramienta de modelado lo asigna al atributo `defaultValue`. La plantilla `generate()`, listada a continuación, opera sobre el valor del atributo `defaultValue`, transformándolo en el valor que se sigue al operador de asignación en la definición de la propiedad objetivo. Esta plantilla `generate()` tiene dos variantes: la primera produce una literal de alguno de los tipos primitivo de Swift a partir de una literal de alguno de los tipos primitivos de UML 2; la segunda produce una literal de alguna de las enumeraciones definidas en el modelo fuente, o una instancia de alguna de las clases definidas en el modelo fuente.

```
1.  [**
2.   * Genera el valor a asignar a una constante o variable a partir de
3.   * una literal. La plantilla decide entre generar una literal de tipo
4.   * cadena de caracteres, entera, real o boolean.
5.   *
6.   * @param aValue
7.   */]
8.  [template private generate(aValue: ValueSpecification) ?
9.                      (aValue.oclIsKindOf(LiteralSpecification))
10.                     post(trim())]
11. [if (aValue.oclIsKindOf(LiteralString))]
12. "[aValue.stringValue()/]"
13. [elseif (aValue.oclIsKindOf(LiteralInteger))]
14. [aValue.integerValue()/]
```

```
15.  [elseif (aValue.oclIsKindOf(LiteralReal))]
16.  [aValue.realValue()/]
17.  [elseif (aValue.oclIsKindOf(LiteralBoolean))]
18.  [aValue.booleanValue()/]
19.  [/if]
20.  [/template]
21.
22.  [**
23.   * Genera el valor a asignar a una constante o variable a partir de
24.   * una instancia. La plantilla decide entre generar una literal de una
25.   * enumeración o una instancia de una clase.
26.   *
27.   * @param aValue
28.   */]
29.  [template private generate(aValue: ValueSpecification) ?
30.                (aValue.oclIsKindOf(InstanceValue)) post(trim())
31.                {anInstanceValue: InstanceValue = aValue.oclAsType(InstanceValue);}]
32.  [if (anInstanceValue.type.oclIsKindOf(Enumeration))]
33.  [anInstanceValue.type.name()/].[anInstanceValue.instance.name/]
34.  [elseif (anInstanceValue.type.oclIsKindOf(Class))]
35.  [anInstanceValue.type.name()/]()
36.  [/if]
37.  [/template]
```

Declaración de operaciones

A las operaciones de clases en Swift las denominamos *operaciones objetivo*; a los elementos de modelado equivalentes en los diagramas de clase los denominamos *operaciones fuente*, y son instancias de la metaclase `Operation`. La plantilla `generateOperationDeclaration()` produce código que declara una operación objetivo, que incluye: el identificador; la lista de parámetros especificados en la operación fuente, si está presente; el tipo del valor de retorno, si está presente; y la palabra clave `override`, si la operación redefine a otra en la superclase.

La plantilla `generateOperationDeclaration()` tiene tres variantes que reciben como parámetro una operación fuente. La plantilla definida en las líneas 9–18 se ejecuta si la operación fuente que recibe como parámetro modela un inicializador, una operación cuyo identificador es `init()` y que asigna valores iniciales a las propiedades de su clase. Esta plantilla toma en cuenta si la operación fuente redefine otras operaciones y coloca la palabra clave `override` de ser así.

La plantilla definida en las líneas 27–38 se ejecuta cuando el diseñador especifica un tipo para el valor de retorno de la operación fuente, por lo que esta plantilla produce el nombre del tipo del valor de retorno. Si la operación fuente redefine al menos una operación en las superclases, la plantilla coloca la palabra clave `override` antes de la declaración de la operación objetivo.

Finalmente, la plantilla definida en las líneas 48–57 se ejecuta si el diseñador no especifica un tipo para el valor de retorno de la operación fuente, por lo que esta plantilla no produce nada seguido de la lista de parámetros de la función objetivo. La plantilla toma en cuenta si la operación fuente redefine otras operaciones y coloca la palabra clave `override` de ser así.

```
1.  [**
2.   * Genera la definición de una operación de inicialización para
3.   * una clase. La plantilla decide si es necesario añadir la palabra
4.   * clave override para indicar que el inicializador redefine a una
5.   * operación en la súper-clase.
6.   *
7.   * @param anOperation
8.   */]
9.  [template private generateOperationDeclaration(anOperation: Operation) ? (anOperation.name.toLower() = 'init')
10.                                                 post(trim())]
11. [if (anOperation.redefinedOperation->isEmpty())]
12. init ([anOperation.ownedParameter->generateParameterList()/]) {
13. }
14. [else]
15. override init ([anOperation.ownedParameter->generateParameterList()/]) {
16. }
17. [/if]
18. [/template]
19.
20. [**
20.  * Genera la definición de una operación que regresa un valor (función).
21.  * La plantilla decide si es necesario añadir la palabra clave override
22.  * para indicar que la operación redefine a una operación en la súper-
23.  * clase.
24.  *
25.  * @param anOperation
26.  */]
27. [template private generateOperationDeclaration(anOperation: Operation) ? (anOperation.type <> null)
28.                                                 post(trim())]
29. [if (anOperation.redefinedOperation->isEmpty())]
30. func [anOperation.name/] ([anOperation.ownedParameter->generateParameterList()/]) -> [anOperation.type.name()/] {
31.   return [anOperation.type.name()/]()
32. }
33. [else]
34. override func [anOperation.name/] ([anOperation.ownedParameter->generateParameterList()/]) -> [anOperation.type.name()/] {
35.   return [anOperation.type.name()/]()
36. }
37. [/if]
38. [/template]
```

```
39.
40.   [**
41.    * Genera la definición de una operación que no regresa un valor
42.    * (procedimiento). La plantilla decide si es necesario añadir la
43.    * palabra clave override para indicar que la operación redefine
44.    * a una operación en la súper-clase.
45.    *
46.    * @param anOperation
47.    */]
48.   [template private generateOperationDeclaration(anOperation: Operation) ? (anOperation.type = null)
49.                                                   post(trim())]
50.   [if (anOperation.redefinedOperation->isEmpty())]
51.   func [anOperation.name/] ([anOperation.ownedParameter->generateParameterList()/]) {
52.   }
53.   [else]
54.   override func [anOperation.name/] ([anOperation.ownedParameter->generateParameterList()/]) {
55.   }
56.   [/if]
57.   [/template]
```

Cuando el diseñador especifica una lista de parámetros para una operación fuente, la herramienta de modelado crea una colección ordenada de instancias de la metaclase `Parameter`, y la almacena en el atributo `ownedParameter` de la operación fuente. Las consultas `isIn()`, `isOut()`, `isInOut()` e `isReturn()`, listadas más adelante, operan sobre una instancia de `Parameter`, verifican el valor de su atributo `direction`, y determinan el tipo de parámetro especificado.

Un parámetro de entrada en la operación fuente se implementa en la operación objetivo mediante un parámetro ordinario con paso por valor; un parámetro de salida en la operación fuente no tiene contraparte en el lenguaje Swift; un parámetro de entrada/salida en la operación fuente se implementa con un parámetro `inout` en la operación objetivo, lo que permite que la operación modifique su valor; finalmente, el parámetro de retorno en la operación fuente no tiene contraparte en Swift, pero es necesario especificar uno para establecer el tipo del valor de retorno de la operación.

La consulta `filterParameters()` recibe un conjunto ordenado de instancias de `Parameter`, y regresa un conjunto ordenado con los mismos elementos exceptuando los parámetros de salida y retorno. Esta consulta opera sobre el atributo `ownedParameter` y excluye aquellos parámetros de la operación fuente que no tienen equivalencia en el lenguaje Swift.

```
1.    [**
2.     * Elimina de un conjunto ordenado de parámetros aquellos que están
3.     * marcados como de salida o de regreso.
```

```
4.      *
5.      * @param parameterSet
6.      */]
7.     [query private filterParameters(parameterSet: OrderedSet(Parameter)): OrderedSet(Parameter) =
8.     parameterSet->reject(aParameter | aParameter.isReturn() or aParameter.isOut())
9.     /]
10.
11.    [**
12.     * Verifica si el parámetro sobre el que se aplica la consulta está
13.     * marcado como entrada.
14.     *
15.     * @param aParameter
16.     */]
17.    [query private isIn(aParameter: Parameter): Boolean =
18.    aParameter.direction = ParameterDirectionKind::_in
19.    /]
20.
21.    [**
22.     * Verifica si el parámetro sobre el que se aplica la consulta está
23.     * marcado como salida.
24.     *
25.     * @param aParameter
26.     */]
27.    [query private isOut(aParameter: Parameter): Boolean =
28.    aParameter.direction = ParameterDirectionKind::out
29.    /]
30.
31.    [**
32.     * Verifica si el parámetro sobre el que se aplica la consulta está
33.     * marcado como entrada/salida.
34.     *
35.     * @param aParameter
36.     */]
37.    [query private isInOut(aParameter: Parameter): Boolean =
38.    aParameter.direction = ParameterDirectionKind::inout
39.    /]
40.
41.    [**
42.     * Verifica si el parámetro sobre el que se aplica la consulta está
43.     * marcado como de regreso.
44.     *
```

```
45.      * @param aParameter
46.      */]
47.     [query private isReturn(aParameter: Parameter): Boolean =
48.     aParameter.direction = ParameterDirectionKind::return
49.     /]
```

La plantilla `generateParameterList()`, listada a continuación, recibe como parámetro un conjunto ordenado de instancias de `Parameter`, e itera sobre los elementos de dicho conjunto para producir una secuencia de declaraciones de parámetros separada por comas.

```
1.      [**
2.       * Genera una lista, posiblemente vacía, de parámetros de una
3.       * operación, separados por comas.
4.       *
5.       * @param parameterSet
6.       */]
7.      [template private generateParameterList(parameterSet: OrderedSet(Parameter)) post(trim())
8.                  {filteredSet: OrderedSet(Parameter) = parameterSet->filterParameters();}]
9.      [for (aParameter: Parameter | filteredSet) separator(',')][aParameter.generateParameterDeclaration()/][/for]
10.     [/template]
```

Un parámetro se declara indicando su identificador seguido de dos puntos seguido del nombre de su tipo de datos. Si la declaración de un parámetro está precedida por el carácter '_' se elimina la posibilidad de utilizar el nombre del parámetro como etiqueta del argumento cuando se invoca la operación objetivo.

La plantilla `generateParameterDeclaration()`, listada más adelante, se invoca por la plantilla descrita anteriormente de forma iterativa hasta producir la declaración de todos los parámetros especificados para una operación fuente. Esta plantilla tiene dos variantes que reciben como parámetro una instancia de `Parameter`, y se ejecutan sí y solo sí dicha instancia tiene un tipo de datos establecido.

La plantilla definida en las líneas 9–15 produce una sentencia que contiene la declaración del parámetro y la asignación del valor por omisión, indicado por el atributo `defaultValue`, lo que aplica únicamente a los parámetros de entrada. La plantilla definida en las líneas 25–33 produce la declaración de un parámetro que no tiene un valor por omisión, y que puede ser de entrada o de entrada/salida.

```
1.      [**
2.       * Genera la declaración de un parámetro individual en la lista de
3.       * parámetros de una función, siempre y cuando tenga un tipo definido,
4.       * esté marcado como entrada y tenga definido un valor por omisión. La
5.       * plantilla genera la asignación del valor por omisión al parámetro.
6.       *
```

```
7.      * @param aParameter
8.      */]
9.     [template private generateParameterDeclaration(aParameter: Parameter) ?
10.                    (aParameter.type <> null and aParameter.defaultValue <> null)
11.                    post(trim())]
12.    [if (aParameter.isIn())]
13.    _ [aParameter.name/]: [aParameter.type.name()/] = [aParameter.defaultValue.generate()/]
14.    [/if]
15.    [/template]
16.
17.    [**
18.     * Genera la declaración de un parámetro individual en la lista de
19.     * parámetros de una función, siempre y cuando tenga un tipo definido
20.     * y no tenga un valor por omisión. La plantilla decide si el parámetro
21.     * está marcado como de entrada o como de entrada/salida.
22.     *
23.     * @param aParameter
24.     */]
25.    [template private generateParameterDeclaration(aParameter: Parameter) ?
26.                    (aParameter.type <> null and aParameter.defaultValue = null)
27.                    post(trim())]
28.    [if (aParameter.isInOut())]
29.    _ [aParameter.name/]: inout [aParameter.type.name()/]
30.    [elseif (aParameter.isIn())]
31.    _ [aParameter.name/]: [aParameter.type.name()/]
32.    [/if]
33.    [/template]
```

3.3.2. Análisis de resultados

A continuación, se presentan evidencias del correcto funcionamiento de la transformación descrita; para cada una de estas evidencias se describen las configuraciones de los elementos de modelado y cómo se implementan en el código. Los segmentos de código expuestos son producto de la ejecución de la transformación sobre algunas de las instancias de `Class` presentes en el diagrama de la Figura 2.

Clases en la jerarquía de la clase `Persona`

En el diagrama de la Figura 2, `Persona` es la clase más abstracta en la jerarquía que categoriza a los individuos que tienen relación con la aerolínea. A continuación, se muestra el código que la plantilla `generateClassDeclaration()` produce para el elemento de modelado utilizado para definir `Persona` en el diagrama; posteriormente se plantean algunas observaciones sobre el proceso de transformación.

```
1.    class Persona {
2.        var nombre: String
3.        var fechaNacimiento: Fecha?
4.        var edad: Int {
5.            get { return Int() }
6.            set {}
7.        }
8.        var sexo: EnumSexo
9.        var numeroPasaporte: Int
10.
11.       init () {
12.       }
13.       init (_ nombre: String,
14.           _ fechaNacimiento: Fecha,
15.           _ sexo: EnumSexo,
16.           _ numeroPasaporte: Int) {
17.       }
18.   }
```

1. Notar que la clase `Persona` en la Figura 2 es abstracta, pero no hay ninguna indicación al respecto en el código producido. Como Swift no permite declarar clases abstractas, la transformación no toma en cuenta ese rasgo del elemento de modelado en el diagrama.
2. El diagrama muestra que las propiedades `nombre`, `sexo` y `numeroPasaporte` de la clase `Persona` tienen multiplicidad uno, es decir, el valor de los atributos `upper` y `lower` es uno para cada una de estas propiedades. La transformación produce, a partir de las propiedades fuente, declaraciones de propiedades objetivo que almacenan un único valor a la vez.
3. Aunque no es evidente en el diagrama de la Figura 2, el valor del atributo `isReadOnly` es falso para todas las propiedades fuente de la clase `Persona`, por lo que la transformación produce declaraciones de propiedades objetivo que son variables.
4. En el diagrama, la multiplicidad de la propiedad fuente `fechaNacimiento` indica que puede tomar un valor o ninguno, lo que se implementa en Swift declarando una propiedad de tipo opcional. Una variable opcional se declara colocando el símbolo "?" a continuación del identificador del tipo de la variable.
5. La propiedad fuente `edad` es derivada, es decir, su valor se calcula a partir de los valores de otras propiedades y no se almacena; se configura asignando el valor verdadero al atributo `isDerived`. Esta propiedad fuente se implementa en Swift declarando una propiedad objetivo calculada, cuyas secciones `get{}` y `set{}` se pueden completar posteriormente.
6. La transformación genera correctamente la definición de las dos variantes de la operación `init()`, así como la lista de parámetros de la segunda operación.

Ahora consideremos el caso de la clase concreta `Sobrecargo` ilustrada en la Figura 2, la cual es subclase de `Tripulante` en forma directa, y de `Empleado` y `Persona` en forma indirecta. A continuación, se muestra el código producido por la transformación para dicho elemento de modelado; posteriormente se plantean algunas observaciones pertinentes sobre el proceso de transformación.

```
1.    class Sobrecargo: Tripulante {
2.        var horasServicio: Int
3.        var numeroCertificacion: String
4.        var listaIdiomas: Set<EnumIdioma> = []
5.
6.        override init () {
7.        }
8.        init (_ nombre: String,
9.            _ fechaNacimiento: Fecha,
10.           _ sexo: EnumSexo,
11.           _ numeroPasaporte: Int,
12.           _ id: Int,
13.           _ numeroSeguridadSocial: Int,
14.           _ salarioBase: Double,
15.           _ cuentaBancaria: CuentaBanco,
16.           _ fechaIngreso: Fecha,
17.           _ horasServicio: Int,
18.           _ numeroCertificacion: String) {
19.       }
20.       func agregarIdioma (_ idioma: EnumIdioma) {
21.       }
22.       func hablaIdioma (_ idioma: EnumIdioma) -> Bool {
23.           return Bool()
24.       }
25.   }
```

1. La transformación identifica la relación de generalización entre `Tripulante` y `Sobrecargo`, por lo que la definición de la clase indica que `Sobrecargo` es subclase de `Tripulante`.
2. En el diagrama de clases, la propiedad fuente `listaIdiomas` tiene las siguientes configuraciones: primero, el límite superior de su multiplicidad es ilimitado ("*"); segundo, los atributos `isUnique` e `isOrdered` tienen los valores verdadero y falso, respectivamente. Lo anterior implica que la propiedad objetivo es una colección no ordenada con un número indeterminado de elementos no repetidos, es decir, un conjunto en lenguaje Swift.

3. La transformación identifica correctamente que la operación fuente `init()` sin parámetros redefine al inicializador de la clase `Tripulante`, y coloca la palabra clave `override` al principio de la definición.
4. La operación fuente `hablaIdioma` tiene un parámetro de retorno tipificado como `Boolean` para establecer el tipo del valor de retorno de la operación. La transformación verifica la existencia de este parámetro y lo consulta para obtener información sobre el tipo del valor de retorno de la función.

Finalmente, consideremos el caso de la clase concreta `Infante`, subclase de la clase abstracta `Pasajero`, en el diagrama de clases. A continuación, se muestra el código producido por la transformación para dicho elemento de modelado; posteriormente se plantean algunas observaciones pertinentes sobre el proceso de transformación.

```
1.    class Infante: Pasajero {
2.        var tutor: Adulto?
3.        let limiteInferiorEdad: Int = 0
4.        let limiteSuperiorEdad: Int = 2
5.
6.    }
```

1. El diagrama de la Figura 2 establece una relación de asociación entre las clases fuente `Infante` y `Adulto`, a través del rol `tutor`; la transformación implementa un vínculo entre las clases objetivo `Infante` y `Adulto` declarando la propiedad `tutor`.
2. De acuerdo con el diagrama, una instancia de la clase fuente `Infante` puede estar vinculada a una instancia de la clase fuente `Adulto` o a ninguna; la transformación implementa este escenario declarando la propiedad `tutor` de tipo opcional.
3. Aunque no es evidente en la Figura 2, el atributo `isReadOnly` de las propiedades fuente `limiteInferiorEdad` y `limiteSuperiorEdad` tiene el valor verdadero, por lo que la transformación produce líneas de código que declaran las propiedades objetivo `limiteInferiorEdad` y `limiteSuperiorEdad` como constantes.
4. Aunque no es evidente en la Figura 2, las propiedades fuente `limiteInferiorEdad` y `limiteSuperiorEdad` tienen establecidos valores por omisión en la herramienta Papyrus. La plantilla `generateConstantDeclaration()` identifica este escenario e interpreta estos valores por omisión como valores iniciales que deben ser asignados a las propiedades objetivo.

Clases producidas en la jerarquía de la clase `Aeronave`

La clase `Aeronave`, mostrada en el diagrama de la Figura 2, es la clase más abstracta de la jerarquía que categoriza a las unidades que componen la flota de la aerolínea. A continuación, se lista el código en Swift producido por la plantilla `generateClassDeclaration()` para el elemento de modelado que define la clase `Aeronave`; posteriormente se plantean algunas observaciones sobre el proceso de transformación.

```
1.    class Aeronave {
2.        var velocidadCrucero: Double
3.        var rango: Double
4.        var mtow: Double
5.        var longitud: Double
6.        var envergadura: Double
7.        var matricula: String
8.        var estado: EnumEstado
9.        var vueloActual: Vuelo?
10.
11.       init () {
12.       }
13.       init (_ velocidadCrucero: Double,
14.           _ rango: Double,
15.           _ mtow: Double,
16.           _ longitud: Double,
17.           _ envergadura: Double,
18.           _ matricula: String,
19.           _ estado: EnumEstado) {
20.       }
21.       func asignarVuelo (_ vuelo: Vuelo) {
22.       }
23.   }
```

1. El hecho de que la clase fuente `Aeronave` esté configurada como abstracta en el diagrama no tiene ningún efecto en el código de la clase objetivo, pues Swift no considera el concepto de clase abstracta.
2. El diagrama de la Figura 2 establece una relación de asociación entre las clases fuente `Aeronave` y `Vuelo`, a través del rol `vueloActual`; la transformación implementa un vínculo entre las clases objetivo `Aeronave` y `Vuelo` declarando la propiedad `vueloActual`.
3. El diagrama de clases indica que toda instancia de alguna subclase concreta de `Aeronave` está vinculada a una instancia de la clase fuente `Vuelo` o a ninguna; la transformación implementa este escenario declarando que la propiedad `vueloActual` es de tipo opcional.

La clase fuente `Aeronave` y sus subclases directas, también abstractas, definen propiedades que aplican a todo tipo de aeronave, pero cuyos valores varían de modelo a modelo. Por ejemplo, una aeronave de pasajeros Boeing 747-400 tiene una velocidad de crucero (propiedad `velocidadCrucero`) de 920 Km/h y un rango (propiedad `rango`) de 11,500 Km, pero una aeronave de pasajeros Boeing 737-900 tiene una velocidad de crucero de 850 Km/h y un rango de 4,300 Km.

A continuación, se lista el código producido por la plantilla `generateClassDeclaration()` para los elementos de modelado que definen las clases `B747_400` y `B737_900` en el diagrama de clases de la Figura 2; posteriormente se plantean algunas observaciones sobre el proceso de transformación.

```
1.      class B747_400: BoeingPasajeros {
2.
3.          init (_ matricula: String,
4.              _ velocidadCrucero: Double = 920.0,
5.              _ rango: Double = 11500.0,
6.              _ mtow: Double = 390100.0,
7.              _ longitud: Double = 70.67,
8.              _ envergadura: Double = 64.44,
9.              _ estado: EnumEstado = EnumEstado.operacional,
10.             _ maxNumeroPasajeros: Int = 408,
11.             _ entretenimientoPersonal: Bool = true) {
12.         }
13.     }
14.
15.     class B737_900: BoeingPasajeros {
16.
17.         init (_ matricula: String,
18.             _ velocidadCrucero: Double = 850.0,
19.             _ rango: Double = 4300.0,
20.             _ mtow: Double = 76900.0,
21.             _ longitud: Double = 42.12,
22.             _ envergadura: Double = 35.8,
23.             _ estado: EnumEstado = EnumEstado.operacional,
24.             _ maxNumeroPasajeros: Int = 188,
25.             _ entretenimientoPersonal: Bool = false) {
26.         }
27.     }
```

1. El diagrama de clases establece la existencia de una relación de generalización entre las clases fuente `B747_400` y `BoeingPasajeros`, y otra entre las clases fuente `B737_900` y `BoeingPasajeros`; estas relaciones se implementan en Swift mediante el mecanismo de herencia, especificando que las clases objetivo `B747_400` y `B737_900` son subclases de la clase objetivo `BoeingPasajeros`.
2. Aunque no es evidente en la Figura 2, los parámetros de las operaciones fuente `init()` tienen valores por omisión establecidos, que la transformación incluye como parte de la declaración de las operaciones objetivo. Estos valores deben ser asignados a las propiedades heredadas por la clase objetivo en el momento en que se construyan instancias de `B747_400` y `B737_900`.

4. Conclusiones

Se presentó al lector una introducción a los conceptos fundamentales de la arquitectura dirigida por modelos y los componentes de esta infraestructura; se proporcionó una descripción básica del metamodelo de UML 2; se discutió a detalle la implementación en Acceleo de una herramienta que produce código en Swift que define clases, propiedades y operaciones a partir de diagramas de clases UML 2; finalmente, se analizaron los resultados del sistema al aplicarlo a un modelo de clases simple. Como trabajo a futuro, se puede analizar qué otras construcciones en Swift relacionadas con la definición de clases se puedan modelar con diagramas de clases UML 2.

Se podría pensar que la aplicación presentada en este reporte es muy simple; que no proporciona un gran beneficio al desarrollador; y que no es posible generar gran parte del código de un sistema a partir de modelos en UML 2. Sin embargo, existen evidencias de lo contrario: en un proyecto del autor [2], se diseñó un perfil de UML 2 que extiende los diagramas de actividades y de máquinas de estado para modelar adecuadamente algoritmos que manipulan tramas de bits; además, se desarrolló una transformación en Acceleo que produce código en lenguaje VHDL, que se puede procesar e implementar en un circuito digital. Los resultados obtenidos durante el desarrollo del proyecto indican que el enfoque de arquitectura dirigida por modelos es provechoso para el diseño de sistemas de hardware digital.

Referencias

[1] Apple Inc. *The Swift Programming Language, Swift 4.2 Edition*, 2018.

[2] Balderas Contreras, Tomás. *Model-Based Design of Digital Hardware Systems for Digital Communications*, Ph.D. Thesis, Instituto Nacional de Astrofísica, Óptica y Electrónica, 2012.

[3] Obeo. *Acceleo/User Guide*, 2018. https://wiki.eclipse.org/Acceleo/User_Guide

[4] Object Management Group. *Model Driven Architecture MDA Guide*, Revisión 2.0, 2014.

[5] Object Management Group. *MOF Model to Text Transformation Language, Versión 1.0*, 2008.

[6] Object Management Group. *OMG® Unified Modeling Language® (OMG UML®), Versión 2.5.1*, 2017.

[7] The Eclipse Foundation, *Papyrus User Guide*, 2018. https://wiki.eclipse.org/Papyrus_User_Guide

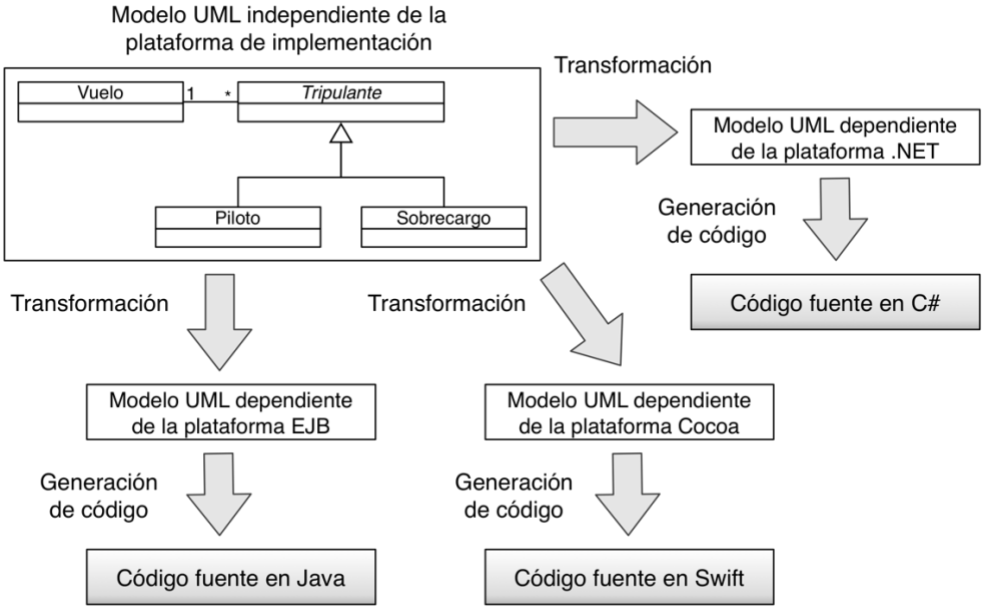

Figura 1. Arquitectura dirigida por modelos para el desarrollo de sistemas de software.

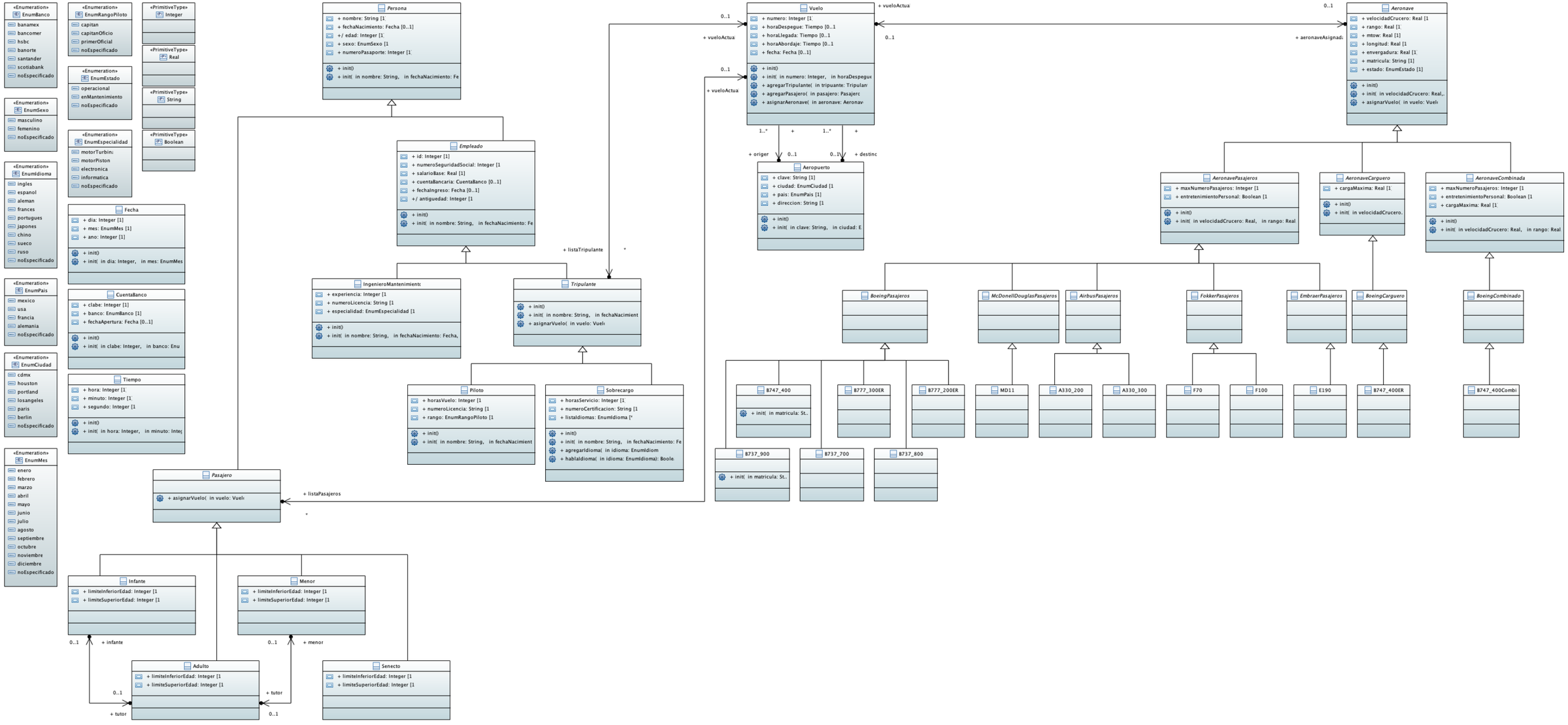

Figura 2. Modelo de clases en UML 2 que describe los elementos que participan en una línea aérea, así como las relaciones entre ellos.

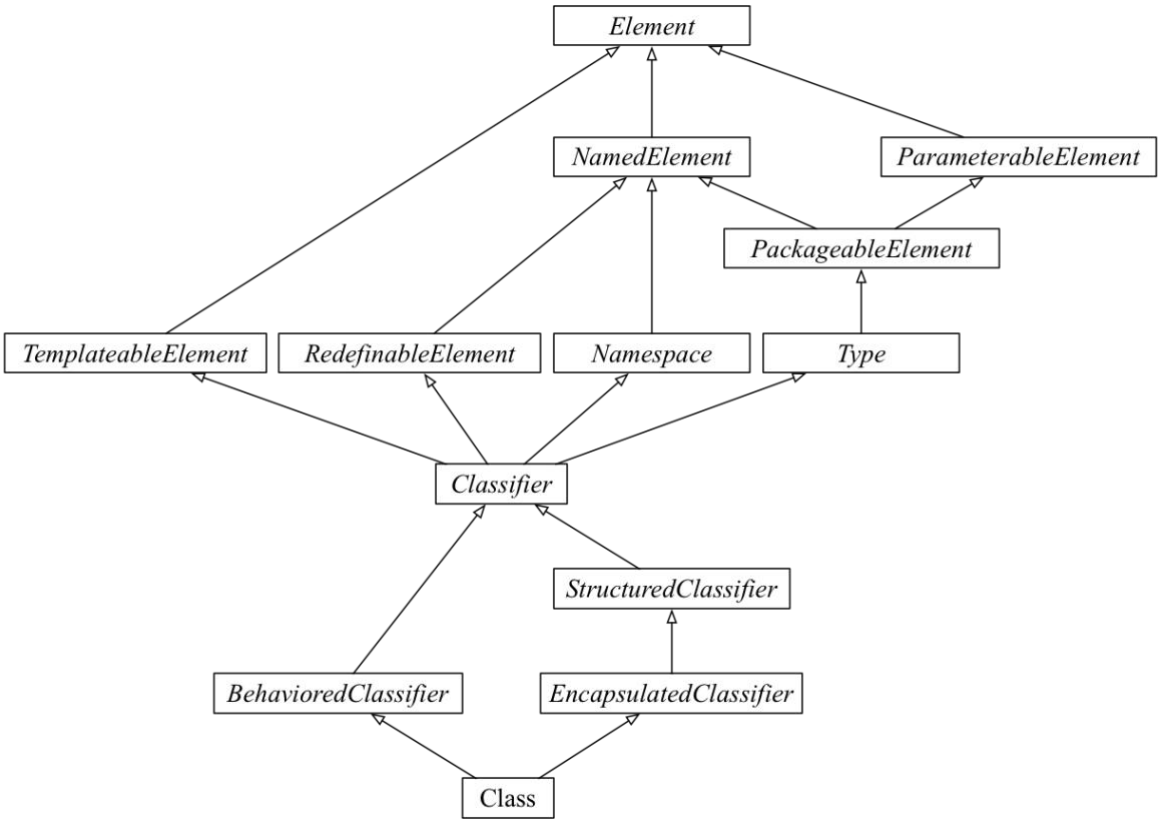

Figura 3. La metaclase concreta `Class` y su jerarquía de metaclases.

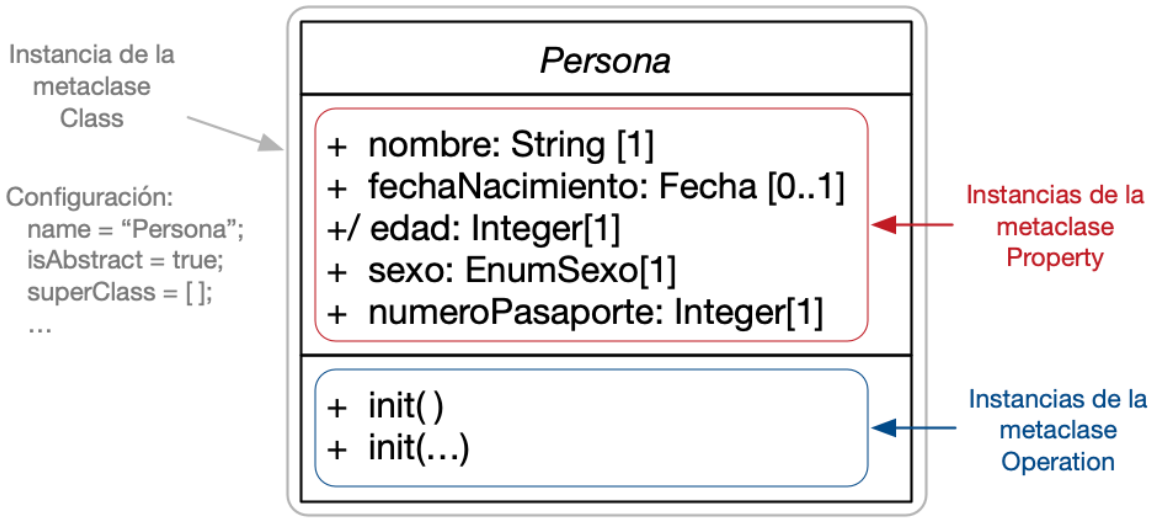

Figura 4. Instancias de las metaclases `Property` y `Operation` contenidas en una instancia de la metaclase `Class`.