

1. INTRODUCCIÓN

Un proyecto de *software* bien documentado tiene más probabilidad de éxito que uno escasamente documentado; por ello se unen esfuerzos en el desarrollo de herramientas que faciliten esta activi- dad, que a pesar de su importancia se le presta poca atención. Swift es un lenguaje que se basa en el paradigma de Programación Orientada a Proto- colo (POP), cada vez más usado por la comunidad Apple. En este sentido, en este trabajo se propone un perfil UML para la documentación en POP. Pre- vio a ello se hace una reflexión del contexto de esta propuesta considerando los tópicos: Documenta- ción del Diseño y POP; luego se plantea el proble- ma, se hace la propuesta del perfil POP y se incor- pora un caso de uso. Se cierra con las conclusio- nes y próximas acciones

1. ANTECEDENTES
2. *Documentación del Diseño*

Todo proceso de desarrollo debe incluir la descrip- ción de la solución propuesta. Cuatro son los con- ceptos que se manejan en este contexto cuando de descripción hablamos **[1].** La **especificación,** la cual tiende a denotar una arquitectura presentada

en un lenguaje formal, pero las especificaciones formales no son prácticas; ni son siempre necesa- rias. La **representación**, que connota un modelo, una abstracción, una interpretación de una cosa separada o diferente de la cosa misma. La **des- cripción**, la cual ha sido replanteada por la comu- nidad del lenguaje de descripción de la arquitectura (ADL). Y la **documentación**, ella denota la crea- ción de un artefacto: es decir, un documento, que puede ser, por supuesto, archivos electrónicos, páginas web o papel. Por lo tanto, documentar una arquitectura de *software* se convierte en una tarea concreta: producir un documento que describa la arquitectura de *software*.

La **arquitectura** es diseño, ella establece restric- ciones en las actividades posteriores, y esas activi- dades deben producir artefactos, diseños y código de grano más fino, que cumplen con la arquitectu- ra. Ella es el vínculo crítico entre el diseño y la in- geniería de requisitos, ya que identifica los princi- pales componentes estructurales de un sistema y las relaciones entre ellos. El resultado del proceso de diseño arquitectónico es un modelo arquitectó- nico que describe cómo se organiza el sistema según la comunicación de sus componentes Es probable que la arquitectura sea costosa **[2].**

La documentación del diseño se usa comúnmente para representar referencias comparables, estable- cer analogías y abstracciones, refinar y evaluar propuestas e interpretar todos estos dibujos a tra- vés de procesos cognitivos. Estas actividades re- fuerzan el papel de la documentación como una herramienta de descubrimiento de diseño, como un medio para refrescar la perspectiva de un proyecto. Es mucho más probable que un proyecto bien do- cumentado esté bien construido que un proyecto pobremente documentado **[3].**

Sin embargo, a pesar de su importancia, según **[4]** la documentación de la arquitectura de *software* tiene la reputación de ser notoriamente mala. Le quita tiempo a la escritura de código. Siempre pa- rece estar desactualizada, por lo general, está es- crita en algún formato de archivo binario patentado que no se puede editar, y además de todo eso,

¡nadie lo lee de todos modos! ¡No es de extrañar que algunas personas llamen SAD (por sus siglas en inglés) a las descripciones de arquitectura de *software*! La esencia es redactar y mantener actua- lizados los resultados de las decisiones arquitectó- nicas para que los *stakeholders* de la arquitectura (las personas que necesitan saber qué trabajo ha- cer) tengan la información que necesitan de forma accesible y sin ambigüedades **[1].**

Esto último es la razón de por qué la notación UML sigue siendo utilizada para documentar las decisio- nes arquitectónicas; en lo que hay que avanzar es en mejorar la eficiencia en la elaboración y actuali- zación de esta documentación. Las herramientas de desarrollo de *software* (CASE) son programas que se utilizan para dar soporte a las actividades del proceso de ingeniería de *software*. Por lo tanto, estas herramientas incluyen dar soporte a la activi- dad de diseño, editores, diccionarios de datos, compiladores, depuradores, herramientas de cons- trucción de sistemas, etc. **[2].** Estas herramientas se pueden combinar dentro de un marco llamado Entorno de Desarrollo Interactivo (IDE), el cual proporciona un conjunto común de herramientas que soportan el proceso de desarrollo lo que las hace más fáciles de usar porque se comunican y operan de manera integrada. Según **[5]** a través de la interfaz de un IDE, un desarrollador o equipo de desarrolladores puede compilar y ejecutar código de manera incremental y administrar los cambios en el código fuente de manera uniforme. Éstos, generalmente están diseñados para integrarse con bibliotecas de control de versiones de terceros, como GitHub y Subversion de Apache. Algunos

entornos pueden admitir el desarrollo basado en modelos (MDD) por lo que en un IDE también se registran modelos y los cambios realizados en ellos.

1. *Programación Orientada a Protocolos y un Perfil de UML*

Los protocolos se utilizan para definir un "modelo de métodos, propiedades y otros requisitos que se adaptan a una tarea o función en particular" **[6].** Swift verifica los problemas de conformidad del protocolo en tiempo de compilación, lo que permite a los desarrolladores descubrir algunos errores fatales en el código, incluso antes de ejecutar el programa. Éstos permiten a los desarrolladores escribir código flexible y extensible en Swift sin tener que comprometer la expresividad del lengua- je. Las extensiones de protocolo se basan en su genialidad **[6].**

Según **[7]** los protocolos permiten agrupar méto- dos, funciones y propiedades similares. Swift le permite especificar estas garantías de interfaz en tipos de *class*, *struct* y *enum*. Solo los tipos de *class* pueden usar clases base y herencia. Una ventaja de los protocolos en Swift es que los obje- tos pueden ajustarse a múltiples protocolos. Al escribir una aplicación de esta manera, el código se vuelve más modular Esto es una característica de calidad que promueve POP.

Piénsese en los protocolos como bloques de cons- trucción de funcionalidad. Cuando se agrega una nueva funcionalidad al conformar un objeto a un protocolo, no se crea un objeto completamente nuevo, eso lleva mucho tiempo. En su lugar, se agregan diferentes bloques de construcción hasta que su objeto esté listo **[7].** La idea de la Progra- mación Orientada a Protocolos (POP) es dejar atrás, en la medida de lo posible, los tipos por refe- rencia (Objetos) y centrarse en los tipos por valor (*Structs*, *Enums*), eliminando varios de los proble- mas que se generan al utilizar Programación Orien- tada a Objetos (POO) **[8].** Ahora bien, dado que POP se ha transformado en un estándar para desarrollos en Swift, surge la necesidad de docu- mentar los diseños que se implementarán bajo este paradigma. Unified Modeling Language (UML), definido por [9], es un lenguaje de propósito gene- ral usado para especificar, visualizar y construir artefactos de sistemas de *software*. Como es un lenguaje de propósito general, la descripción de dominios de aplicación específicos requiere de la definición de un nuevo lenguaje que describe el

metamodelo, utilizando la extensión propiamente de UML a través de un mecanismo denominado Perfiles UML. Los perfiles UML proveen un meca- nismo de extensión genérico para construir mode- los UML en dominios particulares. Están basados en estereotipos, restricciones y valores etiquetados adicionales que son aplicados a los elementos o relaciones de un diagrama **[10].** Es oportuno acla- rar que los perfiles no son una capacidad de exten- sión de primera clase (es decir, no permiten crear nuevos metamodelos), más bien, la intención de los perfiles es brindar un mecanismo directo para adaptar un metamodelo existente con construccio- nes que son específicas para un dominio, platafor- ma o método en particular. Cada una de estas adaptaciones se agrupa en un perfil. Aunque no es posible eliminar ninguna de las restricciones que se aplican a UML cuando se le utiliza, pero si es posi- ble agregar nuevas restricciones que sean especí- ficas del perfil.

1. EL PROBLEMA

Según **[11]** los programadores hacen uso de la documentación en sus actividades diarias, pero a su vez desatienden la creación de la misma, por lo que cada vez se realizan más esfuerzos para la creación de sistemas de autogeneración de docu- mentación.

Como apoyo a esta actividad, se desarrollan he- rramientas de *software* tales como DocC, el compi- lador de documentación *open-source* creado por Apple y la herramienta oficial para la documenta- ción de proyectos de *software* basados en lenguaje de programación Swift **[12].**

Según **[13]** entre los lenguajes de programación más populares de 2022 se encuentran Python, Java, Rust, C++ y Swift, todos estos lenguajes tienen su propia herramienta de generación de documentación, PyDoc, JavaDoc, RustDoc, Do- xygen, DocC en el orden correspondiente. DocC es la herramienta más reciente entre las mencionadas anteriormente, fue anunciada en 2021 y publicada como *open-source* a finales de ese mismo año, aún se encuentra en un estado de desarrollo activo y no iguala el nivel de funcionalidades que ofrecen las herramientas pertenecientes a otros lenguajes de importancia similar. En **[14]**, se indica que una de las funcionalidades que tienen estas herramientas de documentación, excepto DocC, es la generación de diagramas de manera automática mediante el análisis estructural del código compilado. En otras palabras, para que la documentación de los siste-

mas de *software* se considere completa, esta de- biera contener tanto la especificación de las inter- faces ofrecidas como una vista general de la arqui- tectura del sistema **[15];** de esta manera se resalta la importancia de la generación de diagramas de clase en las herramientas de generación de docu- mentación.

No se encontraron estudios donde se hagan análi- sis comparativos entre DocC y otros generadores de documentación, posiblemente debido a lo re- ciente de la herramienta, pero se puede afirmar a partir de lo expuesto que el entorno de generación de documentación Swift, no promueve la completi- tud porque carece de un generador de diagramas.; esto afecta la experiencia del desarrollador y con- secuentemente su reputación. En este sentido, se promueve la mejora a DocC incorporando la capa- cidad de documentar la vista lógica de la arquitec- tura de aquellas aplicaciones desarrolladas con Swift y por ende con el uso de POP, pero para ello, se requiere primero definir un perfil para este para- digma de programación.

1. PERFIL PARA POP

Para la formulación del perfil de UML para POP se hizo un análisis de la guía oficial del lenguaje Swift

1. el cual consistió en su comparación con la guía de especificación UML 2.5 por OMG **[17].**

Como resultado de este análisis se identificaron los **tipos existentes** integrados en el lenguaje Swift, en la sección de referencia del lenguaje, apartado titulado “*Types*”. Esta guía hace mención a los si- guientes tipos: Los tipos existentes se categorizan en tipos con nombre y tipos compuestos. Los tipos con nombre son aquellos a los que se les puede dar un nombre particular al ser definidos, estos incluyen clases, estructuras, enumeraciones, pro- tocolos, *arrays*, diccionarios, *sets*, y todos aquellos tipos que se consideran como primitivos en otros lenguajes (números, caracteres, texto). Los tipos compuestos son tipos sin nombre, estos son las funciones y las tuplas. **[16].**

La Tabla 1 muestra una descripción de cada una de las estructuras de datos mencionadas anterior- mente. Estas descripciones permitieron trazar una base conceptual para luego hacer la comparación entre las estructuras de datos provistas por Swift y las definidas por UML, identificando así cuáles se debían estereotipar

**Tabla I:** Tipos existentes en el lenguaje Swift 5.7

|  |  |
| --- | --- |
| **Nombre** | **Descripción** |
| Clases | “Construcción que nos permite encapsular las propiedades, métodos e inicializadores de un objeto en un solo tipo.” ª  Los objetos se denominan instancias  Pueden relacionarse con otras clases me- diante herencia  Pueden relacionarse con protocolos  Las instancias se comparten mediante refe- rencia, apuntando a la misma dirección en memoria  Es un tipo con nombre  Se puede denominar subclase o superclase dependiendo de su relación con otra clase mediante herencia |
| Estructuras | “Construcción que nos permite encapsular las propiedades, métodos e inicializadores de una instancia en un solo tipo.” ª  No pueden heredar comportamiento ni atri- butos  Pueden relacionarse con protocolos  Las instancias se comparten mediante valor, creando copias  Es un tipo con nombre |
| Enums | “Tipo que define una lista de posibles valores. Adopta muchas características tradicional- mente admitidas solo por clases, como pro- piedades, métodos, inicializadores. Se puede extender para expandir su funcionalidad más allá de su implementación original y puede ajustarse a los protocolos para proporcionar una funcionalidad estándar.” ª |
| Protocolos | “Tipo que define un modelo de métodos, propiedades y otros requisitos que se adaptan a una tarea o funcionalidad en particular. Luego, el protocolo puede ser adoptado por una clase, estructura o enumeración para proporcionar una implementación real de estos requisitos. Cualquier tipo que satisfaga los requisitos de un protocolo se dice que conforma a ese protocolo.” ª |
| Tipos es- tándar (*Int*, *Double*, *String*, etc.) | “Los tipos primitivos de cualquier lenguaje de programación, están implementados por Swift como estructuras por lo que se pueden ex- tender.” ª |
| Colecciones (*array*, *set*, *dictionary*) | “Tipos que actúan como colecciones para guardar valores. Los *array* son colecciones ordenadas de valores, los set son colecciones desordenadas de valores únicos y los diccio- narios son colecciones desordenadas de asociaciones entre un valor y una clave.” ª |
| Funciones | “Tipo que representa los tipos de los métodos, funciones o *closures* y están conformados por parámetros y un tipo de retorno” ª |
| Tuplas | “Las tuplas agrupan múltiples valores en un |

|  |  |
| --- | --- |
| **Nombre** | **Descripción** |
|  | solo valor compuesto. Los valores dentro de una tupla pueden ser de cualquier tipo y no tienen que ser del mismo tipo entre sí” ª |
| Opcionales | “Tipo que manejan la ausencia de un valor (...). Un opcional representa dos posibilida- des: O hay un valor y puede desenvolver el opcional para acceder a ese valor, o no hay ningún valor” ª |

*Nota.* ª Apple (2023)

En la Tabla II se muestran las posibles re- laciones que pueden tener los tipos de la Tabla I.

**Tabla II:** Relaciones existentes en el lenguaje Swift 5.7

|  |  |
| --- | --- |
| **Nombre** | **Descripción** |
| Herencia | “Relación mediante la cual se puede agre- gar o anular funcionalidades mediante composición de clases. La herencia es un comportamiento fundamental que diferencia las clases de otros tipos en Swift” ª |
| *Conformance* | “Relación mediante la cual una clase, es- tructura o enumeración adopta y provee una implementación a los requerimientos de un protocolo” ª |
| Herencia de protocolos | “Relación mediante la cual un protocolo puede heredar uno o más protocolos y puede agregar más requisitos además de los requisitos que hereda” ª |
| Tipos anida- dos | “Relación mediante la cual se define un tipo dentro del contexto de otro tipo. para permi- tir esto Swift permite definir tipos anidados, el cual anida enumeraciones, clases y estructuras compatibles dentro de la defini- ción del otro tipo” ª |
| Extensión | “Relación mediante la cual se le agrega nuevas funcionalidades a clases, estructu- ras, enumeraciones o protocolos ya existen- tes (...). Las extensiones a protocolos sirven para proveer implementaciones por defec- tos a sus métodos y propiedades.” ª |

*Nota.* ª Apple (2023)

Se formuló la suposición que las entidades y rela- ciones a estereotipar son a las que hizo referencia [18], estas son: Protocolos, Estructuras, Enumera- ciones, Extensiones y relaciones de *Conformance*. Desde entonces, el lenguaje Swift ha avanzado a tres versiones mayores [19] y la especificación de UML ha avanzado una versión [20], se hizo nece- sario una revisión de ambos para validar que la

suposición es correcta. Para lograr esto, como la guía de Swift no hace referencia explícita a cuáles de los tipos mencionados son necesarios para po- der hacer uso de POP, se procedió a identificar para las tablas I y II cuál es su estructura equiva- lente definida por UML; lo cual se analizó para ca- da tipo, esta selección se hizo en base a la compa- ración de sus nombres y definiciones. Luego, si se encontraba alguna diferencia entre ellas a nivel conceptual se determinaba que era necesario reali- zar un estereotipo. Este análisis se resume en la tabla III.

**Tabla III:** Comparación entre las estructuras definidas por el lenguaje Swift 5.7 y las definidas por UML 2.5

|  |  |  |  |
| --- | --- | --- | --- |
| **Swift** | **UML**  **Ref.** | **A estereo- tipar** | **Dif. / Obs.** |
| Clases | 11.4.  Clases |  |  |
| Estructuras | 11.4.  Clases | X | Aunque las clases, al igual que las estructuras, sirven para especificar las características y comportamientos de los objetos, estas pueden tener ancestros mediante relaciones de he- rencias, no sopor- tadas por las es- tructuras en Swift. |
| Enums | 10.2.3.3.  Enume- raciones | X | A pesar de que UML define el modelado de las enumeraciones en su apartado  10.2.3.3 estos no son equivalentes, ya que en UML las enumeraciones son consideradas tipos de datos que pue- den almacenar únicamente valores enumerados, mien- tras que las enume- raciones en Swift pueden además de esto almacenar propiedades y métodos. |
| Protocolos | 10.4.  Interfa- ces | X | Existen dos dife- rencias entre los protocolos y las interfaces por los cuales no resultan |

|  |  |  |  |
| --- | --- | --- | --- |
| **Swift** | **UML**  **Ref.** | **A estereo- tipar** | **Dif. / Obs.** |
|  |  |  | estructuras equiva- lentes, estas son:  A diferencia de las enumeraciones en UML, los atributos declarados en los protocolos si deben ser implementados por la estructura que la realiza. Los protocolos pueden ser extendidos para proveer implemen- taciones a sus métodos y atribu- tos, lo cual no es posible con las interfaces |
| Tipos es- tándar | 21.  Tipos Primiti- vos | - | - |
| Tipos de colecciones | 7.5.  Tipos y Multipli- cidad | - | - |
| Funciones | 9.6.  Opera- ciones | - | - |
| Tuplas | 11.5.  *Associa- tions* | - | - |
| Opcionales | 8.6.10.  *Lite- ralNull* | - | - |

En la Tabla IV se muestra un resumen que incluye las relaciones de la Tabla II.

**Tabla IV:** Comparación entre las relaciones definidas por el lenguaje Swift 5.7 y las definidas por UML 2.5

|  |  |  |  |
| --- | --- | --- | --- |
| **Swift** | **UML**  **Ref.** | **A estereoti- par** | **Dif./Obs.** |
| Herencia | 9.9.7. Generali- zación | - | - |
| *Confor- mance* | 10.5.6. *Interfa- ceRealization* | - | - |
| Herencia de proto- colos | 9.9.7. Generali- zación | - | - |

|  |  |  |  |
| --- | --- | --- | --- |
| **Swift** | **UML**  **Ref.** | **A estereoti- par** | **Dif./Obs.** |
| Tipos anidados | - | X | No se  encontró en la guía  UML una descripción de cómo representar la relación de tipos anidados por lo que se deberá estereoti- par. |
| Extensión | 22.3. Estereoti- pos estándar - Refinamiento | X | Aunque el refinamien- to coincide con la  descripción de exten-  sión entre clases, estructuras y enumera- ciones, es necesario refinarlo ya que en  Swift se permite extender protocolos para pro-  veer im- plementa- ciones a  todos los tipos que lo conforman. |

Con base en los análisis de las tablas III y IV se concluyó que, para poder representar un sistema basado en POP, coincidiendo parcialmente con la suposición inicial, es necesario estereotipar los siguientes tipos y relaciones:

* + Estructuras
  + Enumeraciones
  + Protocolos
  + Tipos anidados
  + Extensiones

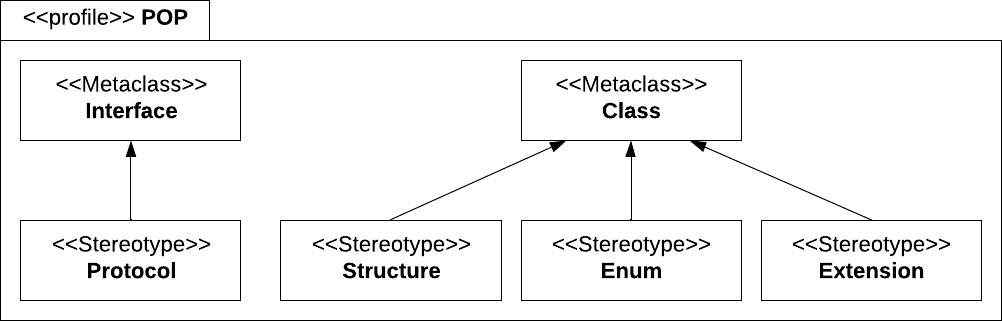
Aunque la relación de tipos anidados no es única a POP, ya que se usa en lenguajes de programación con un enfoque POO como Java, es una relación que también se usa por POP y no está documenta- da por la versión más reciente de UML.

La formulación del perfil incluye los este- reotipos con sus nombres de etiqueta, la metaclase o relación del cual extienden y su descripción. Ver la Tabla V, en ella se usó la misma representación que el OMG usa para los estereotipos estándar de UML 2.5 en su apartado 22.3.

**Tabla V:** Diagrama de Perfil que incluye tipos y relacio- nes estereotipadas para la diagramación de sistemas implementados en Swift 5.7

|  |  |  |
| --- | --- | --- |
| **Nombre** | **Aplica a** | **Descripción** |
| <<nested>> | Composi- ción | Una dependencia de compo- sición que denota una rela- ción de propiedad. Esta rela- ción se usa para representar tipos anidados donde un tipo se declara en el ámbito de otro tipo. El tipo principal declara el tipo interno. |
| <<exten- sion>> | Abstracción | Especifica una relación de refinamiento de una clase, estructura o enumeración. Una relación de extensión entre protocolos provee im- plementaciones a los atribu- tos y métodos de ese protoco- lo. |
| <<Enum>> | Clases | Una clase que soporta, ade- más de propiedades y méto- dos, enumeraciones literales. |
| <<Exten- sion>> | Clases | Una clase del mismo tipo de la clase que extiende. Especi- fíca refinamiento mediante agregación de elementos y atributos. Una extensión relacionada a un protocolo implica proporcionar imple- mentaciones de métodos, y propiedades para los tipos conformes al protocolo. |
| <<Protocol>> | Interfaces | Una interfaz que permite definir propiedades y métodos como requerimientos, además de ser extendido para proveer implementaciones por defecto y conformado mediante la relación *conformance*. |
| <<Structu- re>> | Clases | Una clase que no hereda de otras clases y las instancias se copian por valor, a diferen- cia de la clase tradicional que copia por referencia. |

Una vez formulado el perfil para sistemas basados en POP se propone su representación gráfica, ver **Figura 1.**



**Figura 1:** Diagrama de perfil UML para sistemas codificados bajo el paradigma POP

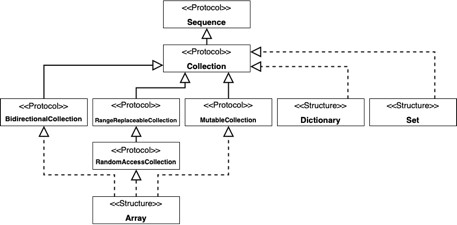
*Nota.* Diagrama de perfil realizado según lo señalado en *UML 2 Toolkit* por OMG (p.39,2004)

1. CASO DE USO DEL PERFIL POP

Para ejemplificar la aplicación del perfil POP se realizó un diagrama de clases con compartimentos suprimidos modelando algunas de las colecciones provistas por la librería estándar de Swift, *Array*, *Set* y *Dictionary*, la cuáles hacen uso extendido de

la composición por protocolos para refinar su com- portamiento **[21]**.

El diagrama realizado aplica algunos de los este- reotipos propuestos en el diagrama de perfil POP, ver **Figura 2**.



**Figura 2:** Modelado de protocolos para las colecciones

*Array*, *Set* y *Dictionary* mediante un diagrama de clases UML aplicando el estereotipo POP

1. CONCLUSIONES

La Programación Orientada a Protocolos se está convirtiendo en un paradigma de programación maduro, y por esto se argumenta la necesidad de formular un modelo formal para el diseño y docu- mentación de sistemas que hagan uso de este.

La documentación de la vista lógica de la arquitec- tura se usa comúnmente para representar referen- cias comparables, establecer analogías y abstrac- ciones, refinar y evaluar propuestas e interpretar todos estos modelos a través de procesos cogniti- vos, de allí su importancia.

El perfil UML para POP propuesto constituye un producto importante que permite cerrar la brecha existente entre las herramientas de diseño destina- das al modelado de objetos y el modelado de pro- tocolos. Además de contribuir al avance del lengua- je Swift, permitiendo desarrollos basados en MDD, como el de la autogeneración de diagramas, en su herramienta de generación de documentación DocC.

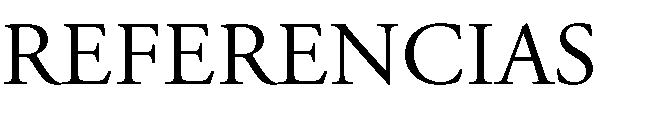
Para su formulación se realizó una revisión docu- mental de la definición formal de UML y la docu- mentación oficial de Swift, comparando y equiva- liendo definiciones, para así concebir estereotipos y valores etiquetados compatibles con el metamode- lo UML 2.5.2, formulados en un diagrama de perfil UML.

Finalmente, se ejemplifica el uso del perfil propues- to, y se aplica al modelado de los protocolos y es- tructuras de las colecciones provistas por la librería estándar de Swift.

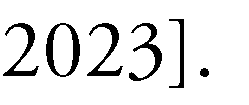
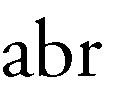
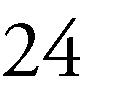
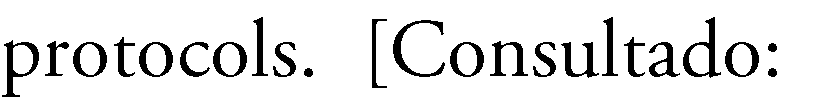
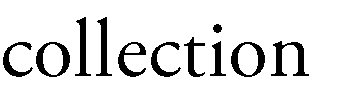
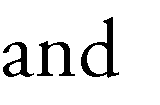
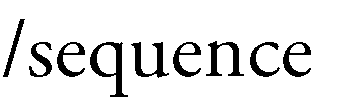
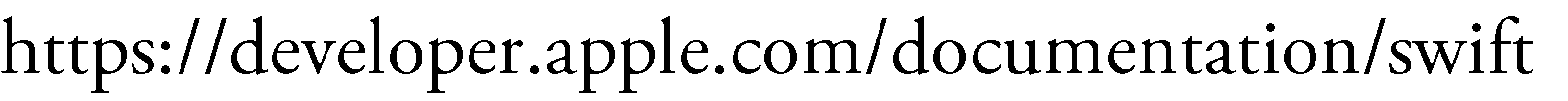
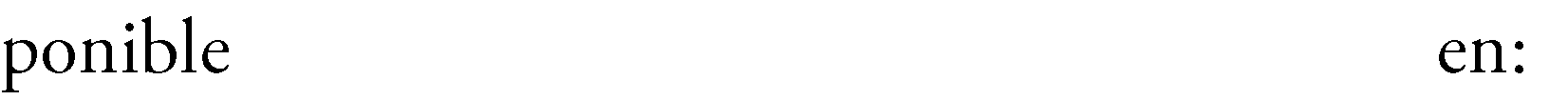
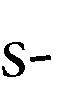
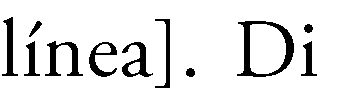
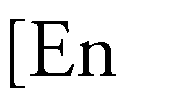
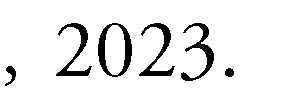
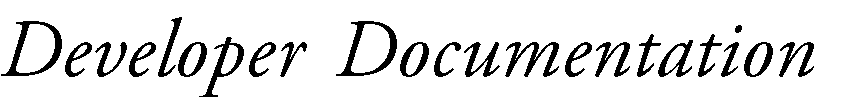
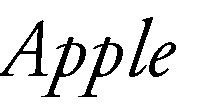
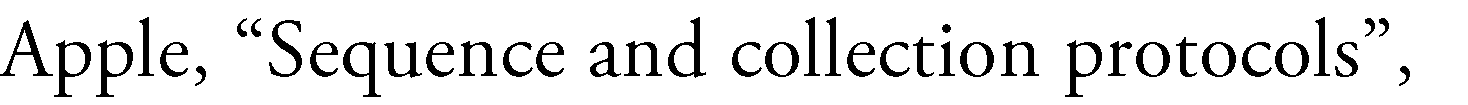
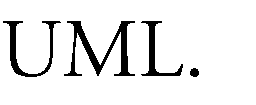
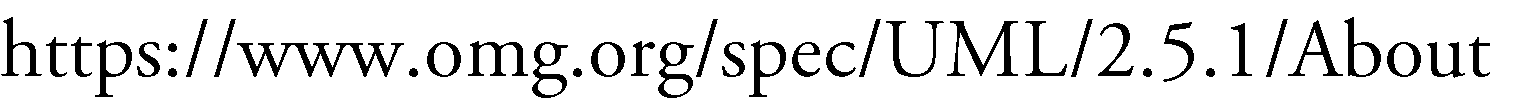
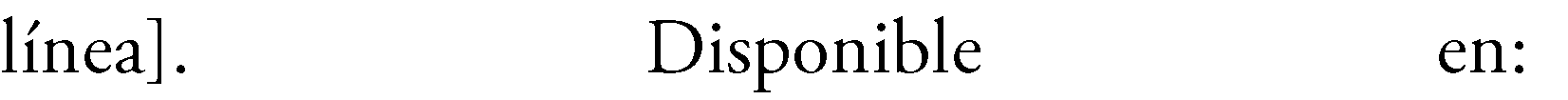
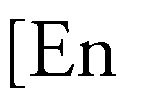
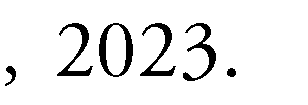
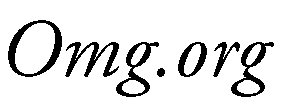
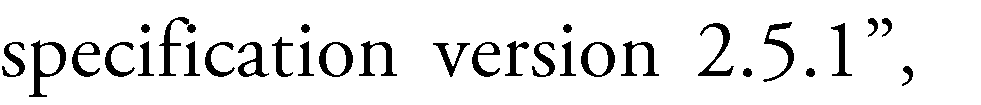
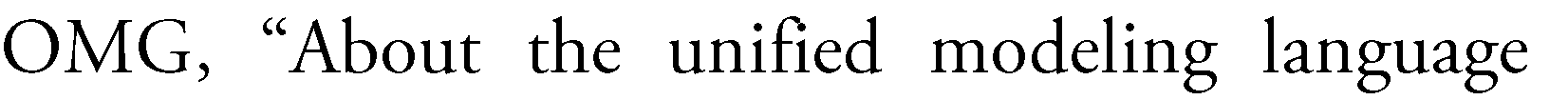
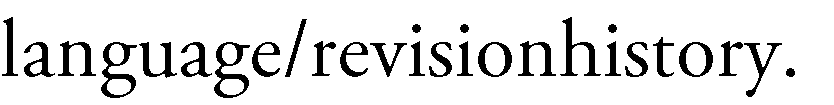
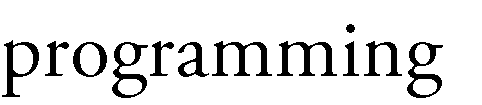
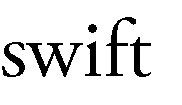
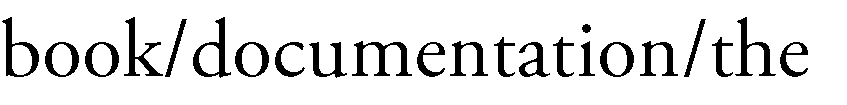
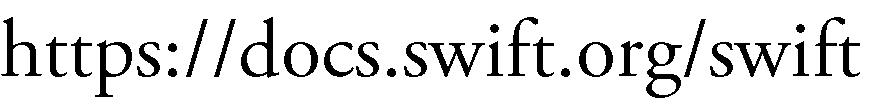
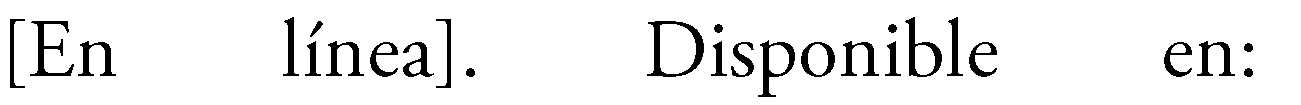
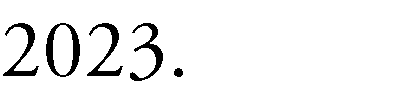
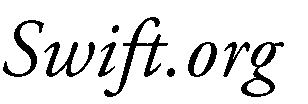
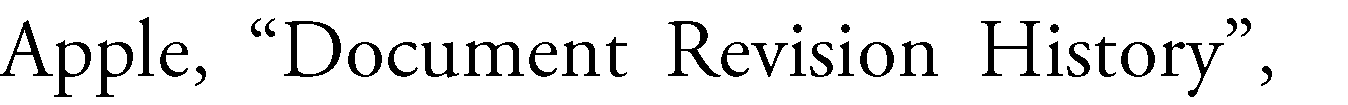
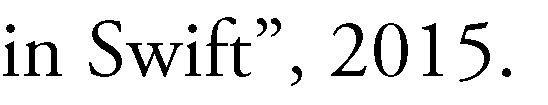
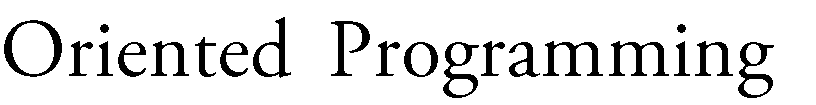
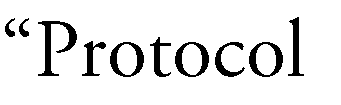
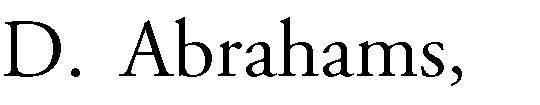
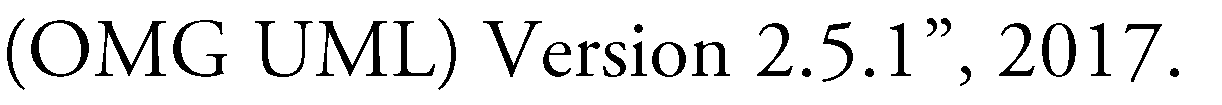
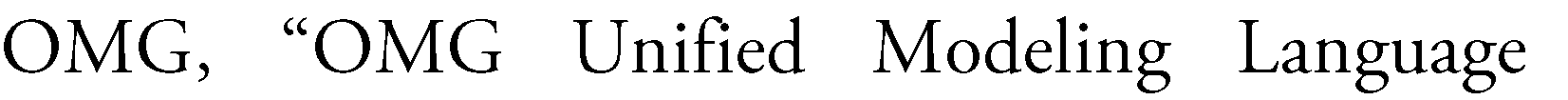
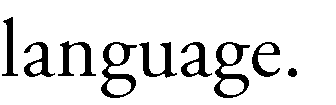
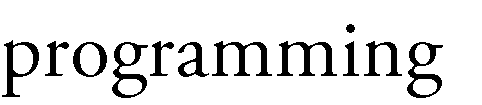
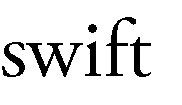
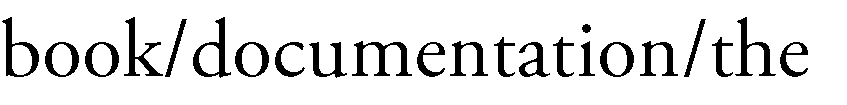
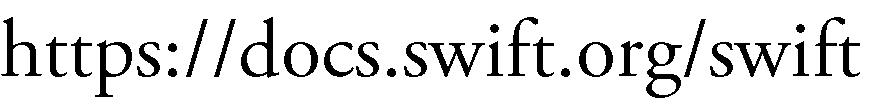
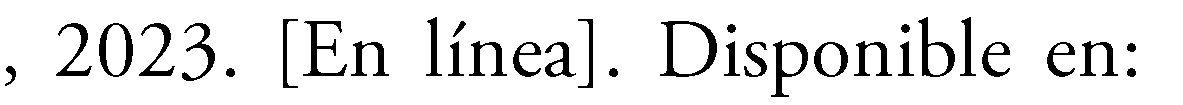
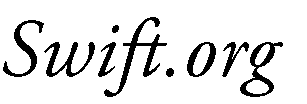
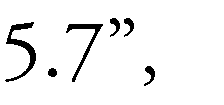
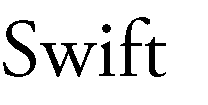
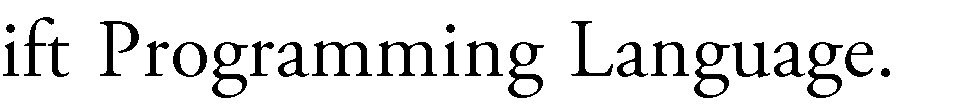
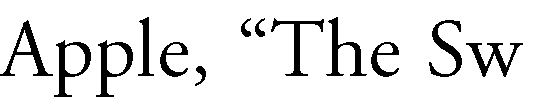
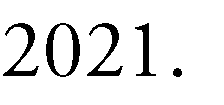
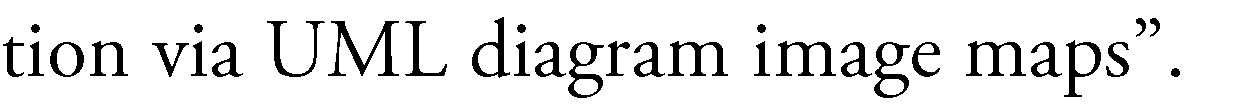
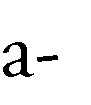
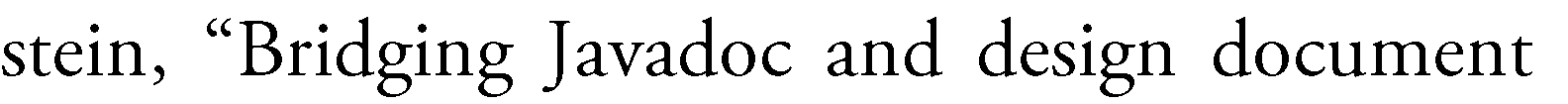
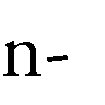
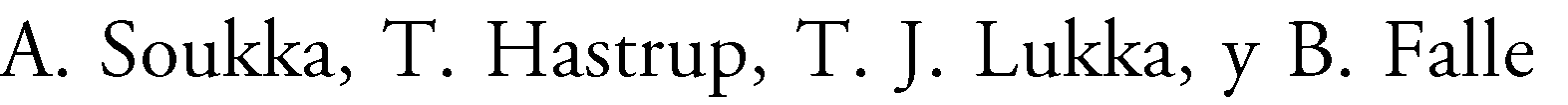
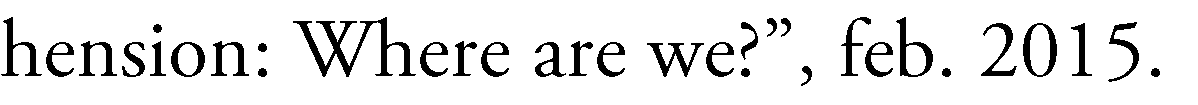
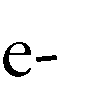
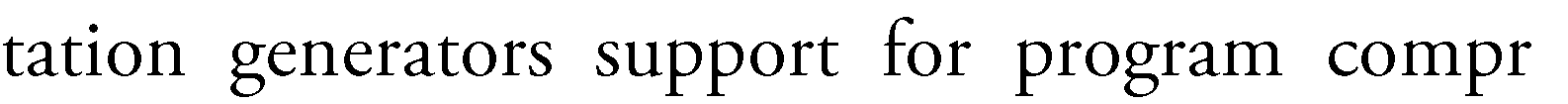
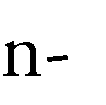
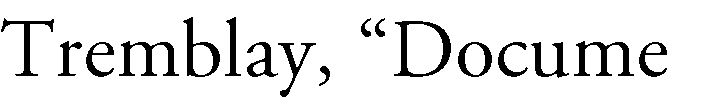
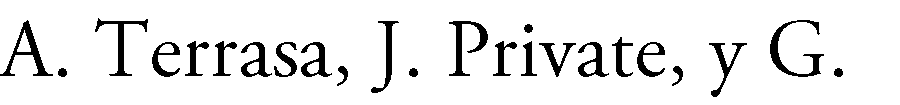
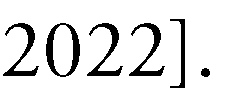
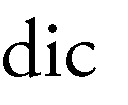
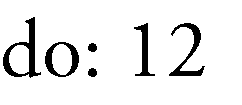
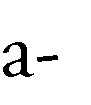
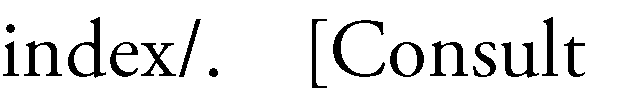
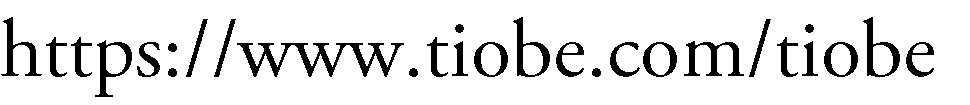
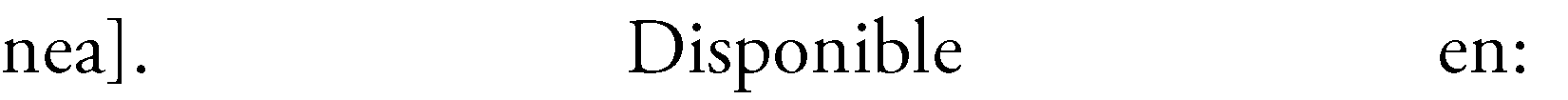
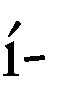
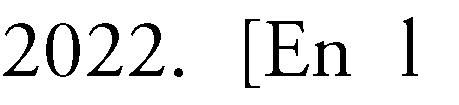
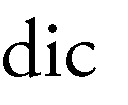
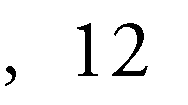
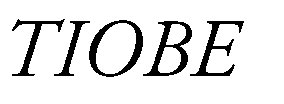
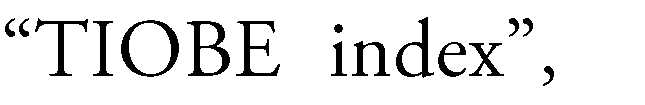
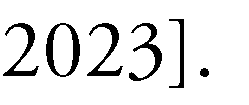
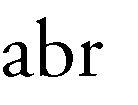
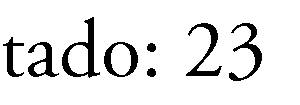
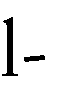
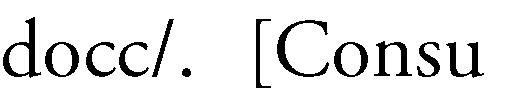
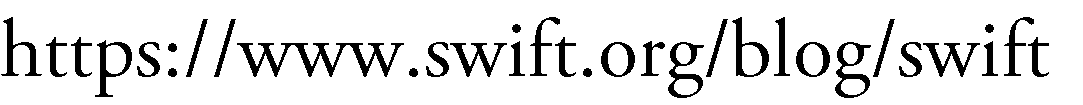
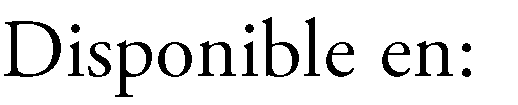
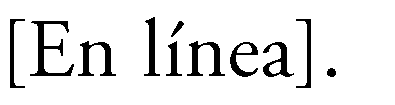
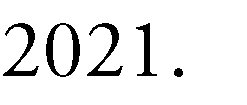
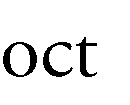
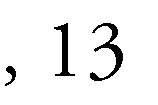
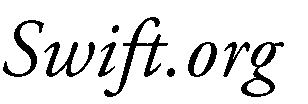
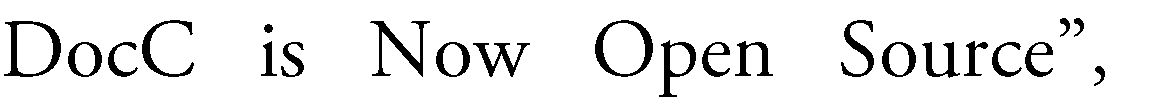
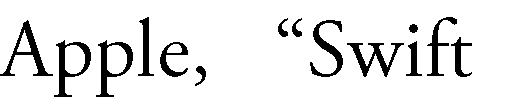
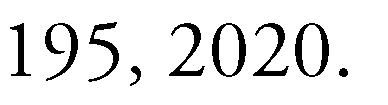
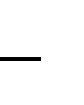
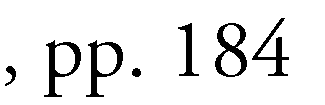
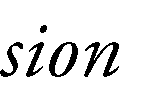
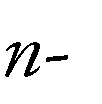
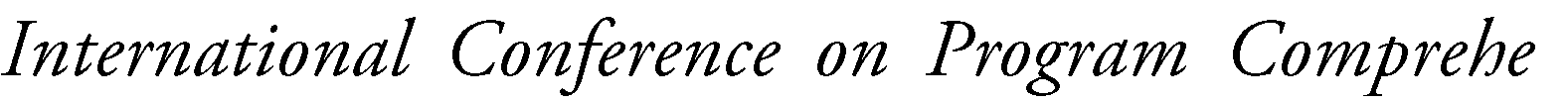
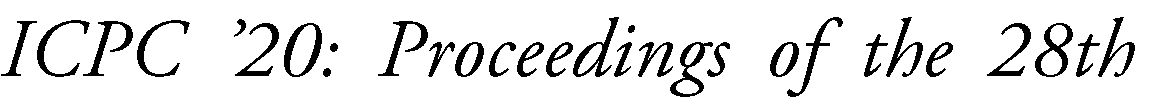
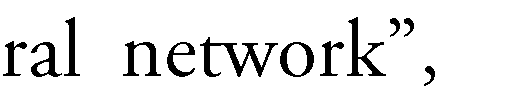
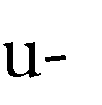
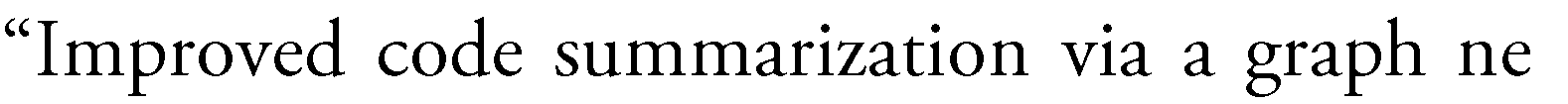
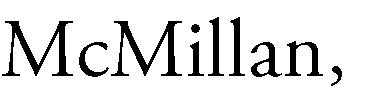
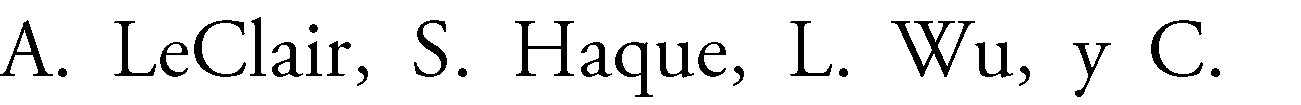
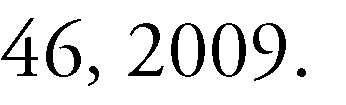
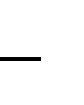
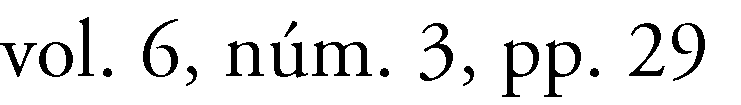
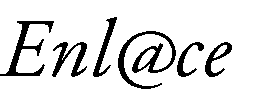
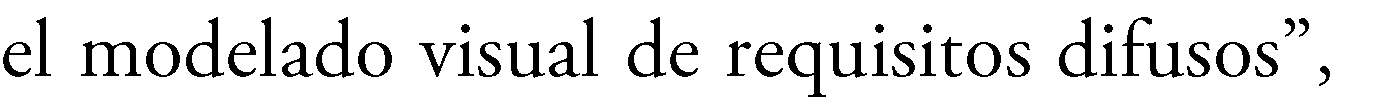
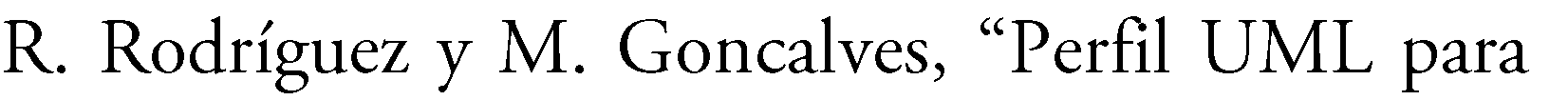
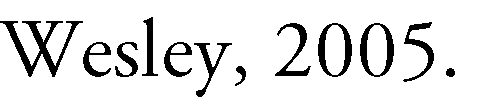
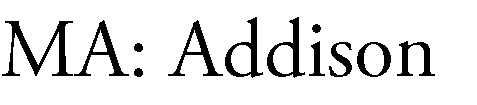
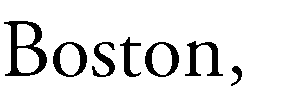
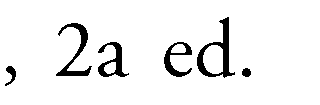
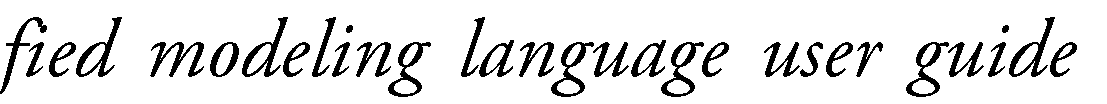
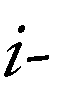
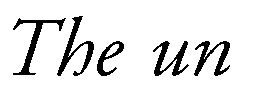
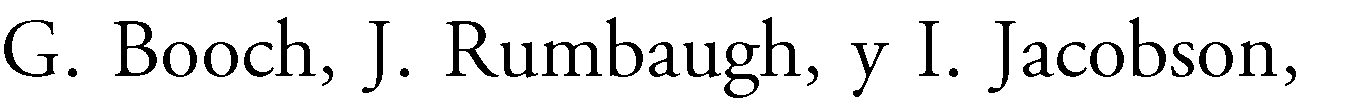
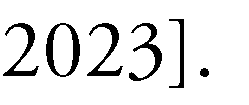
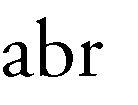
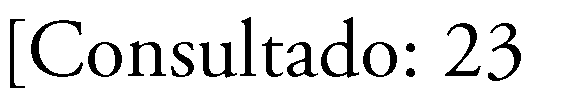
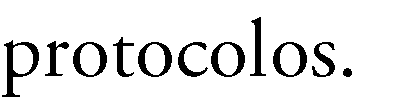
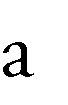
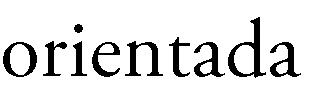
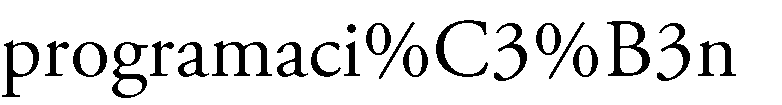
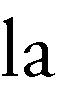
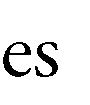
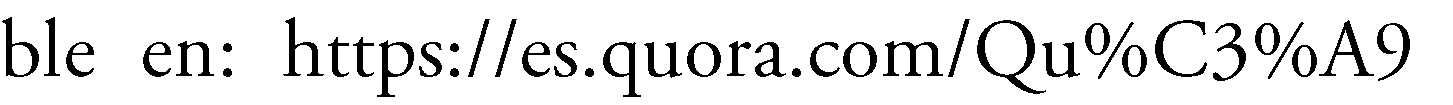
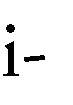
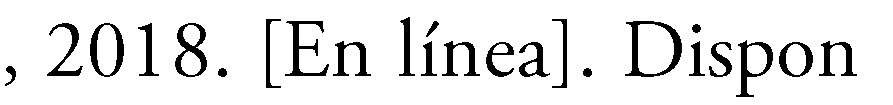
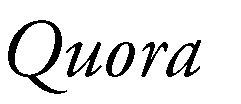
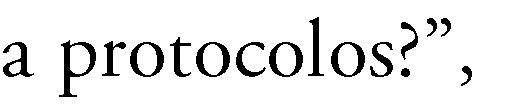
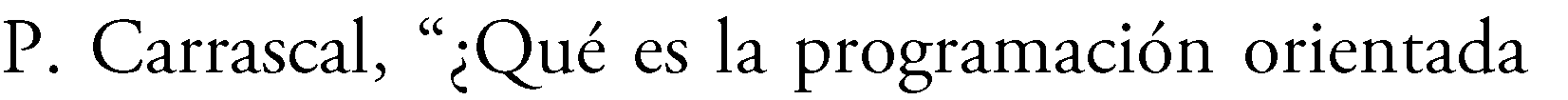
1. ACCIONES FUTURAS

Aún queda camino por recorrer para lograr formali- zar un perfil que exprese todas las bondades del POP. Para que este perfil madure es necesario que la comunidad de programadores de Swift revise su definición, verificando y validando lo aquí propues- to. Específicamente, se recomienda ahondar en las siguientes áreas:

* + Representación de restricciones de protocolos mediante tipos asociados en los diagramas de clase UML.
  + Aplicación del perfil propuesto a diagramas de comportamiento UML.
  + Aplicación del perfil propuesto a diagramas es- tructurales UML.
  + Generación de diagramas como parte del pro- ceso de documentación de sistemas haciendo uso del perfil POP.



|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |



|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |