FACULTAD DE INGENIERIA
UNIVERSIDAD DE BUENOS AIRES

Tesis de Ingeniería en Informática

Estudio de la correspondencia entre modelos UML y código embebido, eficiente y orientado a objetos en lenguaje C

Autor: Jonathan Emanuel Marino (87350)
(jonymarino@gmail.com)

Tutores:

Lic. Rosita Wachenchauzer (rositaw@gmail.com)
Ing. Diego Essaya (dessaya@gmail.com)

Facultad de ingeniería UBA, Buenos Aires
21/09/2019

Resumen

C es un lenguaje de programación desarrollado a principios de la década de 1970, ampliamente usado hasta el día de hoy con gran preponderancia en los sistemas embebidos. El desarrollo de software (en cualquier lenguaje) puede ser potenciado con el uso de modelos y generación automática de código a través de estos lenguajes. Los modelos orientados a objetos son bien representados con UML (por sus siglas en inglés, Unified Modeling Language), un lenguaje de modelado con gran aceptación y popularidad. Por su parte, C es un lenguaje que no contiene conceptos de orientación a objetos en forma nativa.

Esta tesis tiene como primer objetivo estudiar qué modelos orientados a objetos son fácilmente expresables en C cuando es utilizado junto con distintas especificaciones y frameworks que contienen los conceptos de orientación a objetos. Esto puede mostrar a C como un lenguaje apropiado para la enseñanza de la orientación a objetos: un único lenguaje no relacionado con alguna forma de orientación a objetos tomando, de cierta manera, la forma de otros lenguajes y a la vez dejando al descubierto sus costos.

Además, la escritura de código en C bajo estas especificaciones o frameworks es dificultosa. Esta tesis busca, también, analizar en qué medida la experiencia del programador es mejorada al utilizar un generador de código desde UML para producir dicho código.

Por último, el tercer objetivo de esta tesis es encontrar la factibilidad de implementar un generador de código para los frameworks estudiados desde diagramas UML con herramientas actuales.

Agradecimientos

De acuerdo al judaísmo el agradecimiento al Todopoderoso debe ir primero, sin su generosidad nada habría.
Agradezco enormemente a mi esposa Carolina, es imposible lograr objetivos que requieran sacrificio en la vida si nuestra media alma no los valorara, gracias por esperarme tantos días bien entrada la noche para cenar juntos y luego ir a trabajar al otro día temprano, gracias por nuestros hermosos hijos Uriel y David que son el combustible más poderoso para nuestro progreso.
Agradezco a mi Madre por el impresionante esmero y sacrificio que ha hecho para que tengamos una buena educación. Recién cuando uno es padre puede darse cuenta el tremendo desafío que puede ser criar tres hijos tantos años sola. Todavía recuerdo esos tuppers gigantes con el almuerzo y la cena los días que me quedaba todo el día en la facultad. Algo que ahora entiendo es que la autoestima de los hijos está en gran medida depositada en los padres, autoestima que me permite hasta hoy día soñar en grande.
Gracias a mis hermanas Samanta y Johanna por hacer tan agradable la vida en familia, tantos momentos juntos y tantos más por vivir. A mis cuñados Mati y Pato por encargarse de hacerlas felices y encajar tan bien en la familia.
Gracias a mis abuelos Jorge y Lidia que inculcaron en la familia el amor y respeto por el estudio, y que nos ayudaron para que recibiéramos una educación de nivel. Varias veces su casa fue media casa mía para poder estar más cerca de la secundaria o de la universidad y siempre mi abuela me esperó con un plato caliente en su casa. Sé que mi abuelo está contento por este momento en el mundo de la verdad.
Gracias a mi cuñado Federico Somoza, editor de la revista del Hospital Italiano, por la ayuda en la corrección de este escrito y en general por ser

tan buena persona con nosotros.
Gracias a mis suegros Hugo y Lita, si no fuera por su ayuda a la familia sería muy difícil disponer del tiempo para proyectos de cualquier tipo. Gracias por su amor, tolerancia y apertura.
Gracias a mis tutores de tesis que en forma desinteresada se ocupan de los alumnos que deseamos recibirnos con este honor.
Gracias a mi primer jefe, Daniel Litvinov socio de Dhacel SRL, que me permitió desarrollarme y aplicar en los sistemas embebidos los conocimientos que fueron la semilla de esta tesis. Gracias a mi último jefe hasta el momento, Sergio Starkloff socio de Surix SRL, por darme lugar a utilizar el generador de código creado para esta tesis en un ambiente de producción ayudando a profundizar mi conocimiento en el tema.
Gracias a la UBA por la educación formal que he recibido y a la nación Argentina que me la ha dado gratuitamente.
Gracias a cada uno de los familiares, amigos y personas que influencian en mi vida para bien.
Gracias al pueblo judío por preservar su milenaria tradición religiosa de la que me he nutrido diariamente desde los 21 años y me ayuda a ver la vida de forma distinta, con un propósito y, por lo tanto, con más energía.

Por último de nuevo agradezco al Dador constante de la vida ya que todos estos agradecimientos son fundados en Él.

Tabla de contenidos

Abreviaturas

API	**I**nterfaz de **P**rogramación de **A**plicaciones
OO	**O**rientación a **O**bjetos
POO	**P**rogramación **O**rientada a **O**bjetos
LPOO	**L**enguaje de **P**rogramación **O**rientada a **O**bjetos
COO	**C** con características de la **O**rientación a **O**bjetos
LOC	**L**ines **O**f **C**ode: Líneas de código
RTTI	**R**un **T**ime **T**ype **I**nformation: Información de tipo en tiempo de ejecución
UML	**U**nified **M**odel **L**anguage: Lenguaje de modelado unificado

Capítulo 1

Introducción

1.1 Acerca

La programación orientada a objetos en C es posible y es utilizada en la actualidad para todo tipo de aplicaciones a pesar de que C no sea un lenguaje orientado a objetos. Según como se resuelva codificar la orientación a objetos en C, o sea con qué librerías, herramientas y bajo que estándar de C, esa codificación resulta dificultosa. Sin embargo, no por eso debemos desechar a C para programar orientado a objetos. Con las herramientas adecuadas podemos suplir estas dificultades en la codificación orientada a objetos en C.

Esta tesis busca analizar la conveniencia de utilizar un generador de código desde diagramas de clase UML (lenguaje de modelado unificado) como herramienta de ayuda para codificar C orientado a objetos, además de servir como herramienta para facilitar el uso de modelos en el desarrollo.

Para este estudio se han seleccionado distintas especificaciones o frameworks para la codificación de C orientado a objetos, cuyos objetivos van desde los sistemas embebidos altamente restringidos[1] hasta servir de lenguajes de propósito general. Se ha desarrollado, para algunos de ellos, un generador

[1]Nos referimos a restricciones de recursos como memoria ROM y RAM, así como velocidad en ciclos de procesamiento. "Altamente restringido" no es un atributo bien definido, en esta tesis nos referimos a sistemas con menos de 1MB de RAM y ROM. En estos sistemas el lenguaje C es el más utilizado, como veremos.

de código desde diagramas de clase UML analizando la ventaja obtenida al utilizarla.

1.2 Estructura de la tesis

Este es un breve resumen del contenido de cada capítulo. El **Capítulo 2** introduce los conceptos de orientación a objetos y a UML como lenguaje adecuado para representar dichos conceptos. El **Capítulo 3** muestra la necesidad que existe de la programación orientada a objetos en C y da algunos ejemplos de su factibilidad. El **Capítulo 4** introduce distintas especificaciones y frameworks para codificar C orientado a objetos, los conceptos que contienen y sus dificultades en la codificación. Se propone una representación en UML para el código bajo dichas especificaciones o frameworks, representación que está desprovista de dichas dificultades. El **Capítulo 5** introduce herramientas para transformar modelos en código.El **Capítulo 6** muestra la factibilidad de implementar un generador de código desde diagramas UML para los frameworks analizados. El **Capítulo 7** habla acerca de las conclusiones extraídas de esta tesis y trabajo futuro.

Capítulo 2

Programación orientada a objetos y UML

2.1 Orientación a Objetos

2.1.1 Introducción

La mayoría del software contemporáneo es desarrollado bajo el enfoque de la orientación a objetos. Bajo este enfoque la unidad básica de construcción de todos los sistemas de software es el objeto. Un objeto busca representar un sujeto dentro del dominio del problema o la solución; las clases son una descripción general de un objeto y son los generadores de objetos con esa descripción. Cada objeto tiene identidad, estado y comportamiento. Identidad implica que puede distinguirse de los demás objetos y asignársele un nombre en el código. Estado implica que tiene datos asociados con él, que pueden ir cambiando durante la vida del objeto. Comportamiento implica la capacidad de un objeto de hacer cosas y de operar sobre otros objetos y variables incluyendolo a él mismo, la forma de operar sobre los objetos es mediante el envío de mensajes a los mismos. En este enfoque, llamado paradigma, se busca identificar los objetos del dominio del problema y se busca distribuir la responsabilidad de la solución en dichos objetos. Las virtudes de la programación orientada a objetos son por demás

conocidas: modularidad, flexibilidad, extensibilidad, reutilización de código. La orientación a objetos ha sido la aplicación de los buenos principios de programación enseñados con anterioridad a su existencia (Douglass 2010). El enfoque de la orientación a objetos ha demostrado ser valioso en la comprensión de problemas y su solución en todo tipo de dominios de problemas, así también como grados de tamaño y complejidad. Además, la mayoría de los lenguajes contemporáneos están orientados a objetos de alguna manera, lo que proporciona una causa mayor para ver el mundo en términos de objetos. Fuera de esta definición simplista, definir exactamente qué es la orientación a objetos es una tarea muy difícil. Algunos consideran como esencial lo que para otros no lo es. Alrededor de la orientación a objetos distintos conceptos han sido elaborados, y cada implementación a tomado un subconjunto de aquellos (Hendrickx 2004).

2.1.2 Conceptos

Stroutstrup afirma "El soporte básico que necesita un programador para escribir programas orientados a objetos consiste en un mecanismo de clase con herencia y un mecanismo que permite que las llamadas de funciones miembro dependan del tipo de un objeto (en los casos en que el tipo es desconocido en tiempo de compilación)". (Stroustrup 1991) De aquí se desprenden los tres conceptos más esenciales de la orientación a objetos: Encapsulación, herencia y polimorfismo. Para obtener mayor claridad de los frameworks que estudiaremos introduciremos primero estos conceptos básicos. A continuación introduciremos otros conceptos implementados por algunos de ellos. La descripción dada de esos conceptos no es de ninguna manera completa sino introductoria.

2.1.3 Encapsulación

La encapsulación y la ocultación de datos es uno de lo conceptos más importantes de la POO. La misma se consigue a través de la definición de clases en los LPOO. Las clases son el bloque de construcción más importante en la mayoría de los sistemas orientados a objeto (Grady Booch

1998). El implementador de la clase define los tipos de datos de la misma y la forma de interactuar con ellos mediante una interfaz bien definida, puede haber datos totalmente accesibles por el usuario y, por el otro lado, otros totalmente invisibles al mismo.

2.1.3.1 Objetos y clases

Un objeto es descrito por una clase, donde la clase describe un tipo. Un objeto es una instancia de tal tipo. A menudo un objeto se resume como una identidad, un estado, y comportamiento (Booch 1991). El estado de un objeto está determinado por los valores de sus datos, una clase define los tipos de datos de la misma y un objeto contiene valores para esos datos. El comportamiento de un objeto está definido por la interfaz definida a su vez en la clase. Una identidad puede ser cualquier identificador, a menudo referido como una referencia al mismo.

2.1.3.2 Encapsulación fuerte

Si una clase sólo permite acceder a sus datos a través de su interfaz, esto se llama encapsulación fuerte. Ciertas implementaciones (tanto lenguajes como en nuestro caso frameworks) de POO pueden obligar la encapsulación fuerte.

2.1.4 Herencia

La herencia es la capacidad que tiene una clase para extender a otra. Se dice que una clase B extiende otra clase A y, por lo tanto, hereda todas las propiedades de la clase A, eso significa que hereda los datos y la interfaz de la clase A. La herencia es una relación importante entre las clases y es una de las herramientas más utilizadas para la reutilización de código. Las implementaciones de POO pueden soportar tanto **herencia simple** como **herencia múltiple**. Herencia simple significa que una clase sólo puede heredar a lo sumo de otra más. Herencia múltiple significa que una clase puede heredar de una o más clases. La herencia múltiple es bastante

criticada por la complejidad que puede traer en los diseños y la complejidad de resolución del problema del diamante (Estrada 2007).

2.1.4.1 Clase base

Una clase base es una clase que no hereda de ningún otra.

2.1.4.2 Clase derivada o hija

Una clase derivada es aquella que hereda de una o varias clases, que pueden ser a su vez derivadas o base. Se dice que una clase derivada o hija hereda de una clase padre.

2.1.5 Polimorfismo

Es la capacidad de que una referencia a un tipo tome diferentes formas en tiempo de ejecución: el envío de un mensaje a un objeto de cierto tipo (estático) puede invocar diferentes comportamientos según el tipo de tiempo de ejecución (o dinámico).

2.1.5.1 Mensajes y métodos

Un método es la definición del comportamiento de un objeto ante cierto mensaje en cierto contexto, en caso de que una clase padre ya contenga definición de dicho método, una clase hija puede redefinir dicho comportamiento. El usuario de un objeto envía un mensaje al mismo y el objeto puede ejecutar un método ante tal mensaje.

2.1.5.2 Reenvío de mensajes y delegados

Cuando un objeto no posee un método para responder a cierto mensaje, si es que soporta el reenvío de mensajes, puede reenviar el mensaje a otro objeto

que sí sepa responder a dicho mensaje, dicho objeto es un delegado.

2.1.5.3 Métodos abstractos

Los métodos polimórficos definidos por una clase que no define su implementación son llamados métodos abstractos. La idea de un método abstracto es que las clases derivadas definan su implementación. Una clase con métodos abstractos es llamada una clase abstracta y no puede generar instancias de la misma.

Hasta aquí introducimos los conceptos que identificamos como esenciales de la POO. Los siguientes conceptos, sin embargo, están relacionados con la orientación a objetos y son implementados por los frameworks que estudiaremos. Algunos de ellos son considerados como esenciales a la orientación a objetos por algunos gurúes, pero en la práctica son más dependientes de la implementación (Hendrickx 2004).

2.1.6 Interfaces

Una interfaz consiste en un listado de declaración de métodos carentes de implementación. Una clase se dice que realiza una interfaz e incluye a sus métodos como propios dándoles una implementación a los mismos.

2.1.7 Mixin

Los mixins son una mezcla entre clases e interfaces. Definen y pueden implementar nuevos métodos, e incluso pueden poseer datos miembros pero no pueden ser instanciados. Lo único que puede hacerse con un Mixin es que sea heredado por una clase. Puede verse a un Mixin como una interfaz pero que define una implementación.

2.1.8 Metaclases

Las instancias de las clases son objetos. Una metaclase es una clase cuyas instancias son clases. Esto permite crear clases en tiempo de ejecución.

2.1.9 Manejo de excepciones

Un mecanismo para anticipar y manejar excepciones. Se produce una excepción cuando algo no sale como se pretendía. Un mecanismo para un fácil manejo de excepciones permite al desarrollador manejar una excepción elegantemente. Permite separar el código que debería ejecutarse sin problemas de las excepciones y resolver las excepciones en el contexto donde pueden tratarse. Si bien puede haber sistemas de excepciones basadas en un sistema de objetos en C específico a un framework, también existen librerías que pueden utilizarse bajo cualquier especificación o framework.[1]

2.1.10 Recolector de basura

Un sistema que libera memoria no utilizada en el programa durante la ejecución del mismo. Esto libera al desarrollador de tener que manejar la gestión de memoria manualmente. De nuevo existen librerías que son independientes de un framework de orientación a objetos.[2]

2.1.11 Introspección de tiempo de ejecución

La capacidad de realizar una introspección de un objeto en tiempo de ejecución. Esto permite al desarrollador consultar el objeto y solicitar información de tipo de tiempo de ejecución (RTTI). Por este mecanismo puede obtenerse el tipo de un objeto, el nombre del tipo y cuáles son sus clases padres.

[1] Ver por ejemplo Exception y cexception
[2] Ver por ejemplo GCBoehm y TinyGC

2.1.12 Reflexión

La reflexión se define por Peter W. Madany, Nayeem Islam, Panos Kougiouris, y Roy H. Campbell como "la capacidad de un sistema de ejecución de objetos programados para hacer que los atributos de los objetos sean en sí mismos objeto de cálculo" (Peter W. Madany 1992). En otras palabras, la capacidad de obtener los miembros de los objetos y otra información estructural en tiempo de ejecución. La introspección de tiempo de ejecución puede considerarse un nivel bajo de reflexión (Hendrickx 2004).

2.1.12.1 Programación clave-valor (key-value programming)

La codificación de valor-clave es un mecanismo para acceder a las propiedades de un objeto indirectamente, utilizando cadenas para identificar propiedades, en lugar de invocar un método de acceso o acceder a ellas directamente a través de variables de instancia. (**???**).

2.1.13 Duck Typing

Una implementación con capacidad de Duck typing permite enviar cualquier mensaje a cualquier objeto, y la implementación sólo se preocupa de que el objeto pueda aceptar el mensaje, no requiere que el objeto sea de un tipo particular, como hacen Java y C ++ (Eckel s. f.). En cambio duck typing sigue la filosofía de "[si] camina como un pato, nada como un pato y suena como un pato, a esa ave yo la llamo un pato.",[3] lo que significa que los tipos no son lo más importantes sino el comportamiento (métodos) de los objetos.

2.1.14 Genericidad

Existen dos acepciones del término que nos interesan:

[3]Davis, Robin S. "_Who's Sitting on Your Nest Egg?" página 7.

2.1.14.1 En lenguajes fuertemente tipados

Una definición simple de un lenguaje fuertemente tipado es que las funciones escritas para cierto tipo sólo pueden ser invocadas con instancias de esos tipos (u otros que hereden de ellos). Por lo que por cada tipo no relacionado con el otro obligatoriamente hay que escribir otra función o método aunque las funciones hagan prácticamente lo mismo. La genericidad permite escribir funciones independientes del tipo que luego en tiempo de compilación se generan las distintas versiones de la función para los distintos tipos con las que se lo usa.

2.1.14.2 En lenguajes débilmente tipados con funciones genéricas

En lenguajes débilmente tipados pueden definirse funciones genéricas con múltiples implementaciones que difieren en los tipos de sus argumentos. En tiempo de ejecución, al llamar a tales funciones, es ejecutada la implementación de la función que mejor coincida con los tipos utilizados para la llamada (el nivel de la herencia es contemplado en la elección). Si la función a ejecutar depende de varios de sus argumentos, esto se llama despacho múltiple (multiple dispatch), si sólo depende del primer argumento se llama despacho único (single dispatch). La diferencia entre una función con despacho único y un método polimórfico es que la primera no necesita ser definida como miembro de la clase.

2.2 UML y la Programación Orientada a Objetos

2.2.1 Qué es UML

El Lenguaje de modelado unificado (UML) es un lenguaje gráfico para especificar, construir y documentar los artefactos de software. Un artefacto de software es cualquier resultado tangible en el proceso de desarrollo de software. UML provee un lenguaje común para describir las partes

importantes de un sistema desde un enfoque conceptual. Puede describir desde procesos de negocio y funciones de un sistema así como cosas concretas como clases escritas en un lenguaje de programación específico y componentes reutilizables de software.
La notación de UML está especificada por el metamodelo de UML. En esta tesis usaremos la versión de metamodelo 2.5.1 (OMG 2017).
La mayoría del contenido de las próximas secciones está basado en el libro "The Unified Modeling Language User Guide" de los autores de UML (Grady Booch, James Rumbaugh e Ivar Jacobson) (Grady Booch 1998).

2.2.2 Ventajas del uso de UML para todos los LPOO

Un lenguaje consiste en un vocabulario y reglas para utilizar ese vocabulario. Un lenguaje se utiliza para comunicar. Un lenguaje de modelado se utiliza para comunicar modelos. Estos modelos son representaciones conceptuales o físicas de un sistema.
Muchos programadores están acostumbrados a codificar inmediatamente después de haber pensado la solución al problema que están resolviendo (sino antes). A veces el texto es la mejor forma de representar algo, como es el caso en fórmulas matemáticas y algoritmos. En tales casos, los programadores siguen creando un modelo mental que luego es implementado en el código. También podría crear un borrador de su visión mental, pero este enfoque tiene varios inconvenientes: Primero porque comunicar estos modelos a otros es una tarea difícil y propensa a errores. El desarrollador o los desarrolladores crean un lenguaje para poder expresar su modelo y quienes no sean introducidos en ese lenguaje personalizado no podrán o les será difícil entenderlo.

Segundo, hay algunos aspectos de los sistemas de software que no pueden ser directamente comprendidos a través de tan solo mirar el código. Por ejemplo, extraer el significado de cierta jerarquía de herencia de clases desde el código implicaría inferirlo a partir de varios archivos mientras que a con un modelo podría captarse en forma directa. Si el programador que escribió el

código nunca dejó un modelo que documente lo que estaba en su cabeza como solución al problema, esa información puede perderse o llegar a reproducirse parcialmente si ese programador no estuviese más.

2.2.3 UML es un lenguaje para construir

Si bien UML no es un lenguaje de programación visual, sus modelos pueden ser conectados a una variedad de lenguajes de programación. Eso significa que es posible mapear modelos en UML a distintos Lenguajes de Programación Orientados a Objetos como JAVA o C++. Las cosas que son mejor expresadas visualmente son hechas en UML, mientras que las cosas que son mejor expresadas textualmente son hechas en el código. Este mapeo permite la generación de código desde modelos UML, esto es llamado ingeniería directa. También es posible realizar ingeniería inversa: generar modelos UML a partir de código. Como dijimos, el código puede no contener toda la información del modelo por lo tanto lo que no se encuentre en el texto en forma desambiguada no podrá regenerar el modelo a partir del cual se creó el código.

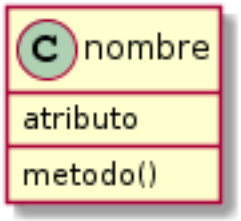
Generar código desde modelos es llamado ingeniería directa, y generar modelos desde código ingeniería inversa. La combinación de los dos da como resultado la ingeniería circular (en inglés round-trip engineering), lo que implica poder trabajar tanto en los gráficos como en el código mientras las herramientas se encargan de mantener la consistencia entre ambos. En esta tesis buscaremos demostrar que la ingeniería directa también es posible para C a pesar de no ser un LPOO. Bajo ese supuesto es de esperar que la ingeniería inversa también sea posible.

2.2.4 El lenguaje de UML

Ahora introduciremos al lenguaje de UML. De ninguna manera buscamos dar una definición cabal del mismo sino tan solo introducir los elementos que utilizaremos para dar una representación en UML a código C escrito bajo las distintas especificaciones y frameworks que estudiaremos.

2.2.4.1 Clases

UML provee una representación gráfica de una clase. La siguiente figura nos muestra tal representación.



Esta notación permite visualizar una abstracción, el concepto de clase, independiente de un lenguaje de programación específico. El mismo enfatiza en tres bloques las partes más importantes de la clase: su nombre, sus atributos y sus métodos.

2.2.4.1.1 Nombres El nombre de una clase debe ser único en el paquete en el cual es contenido (los paquetes se introducirán luego). Los nombres pueden ser cualquier texto incluyendo letras, números y signos de puntuación (excepto el signo :). Esta flexibilidad en el nombre puede no ser soportada por todo lenguaje de programación.

2.2.4.1.2 Atributos El concepto de atributo en UML es equivalente al de cualquier LPOO . Un atributo es una propiedad de una clase que describe un rango de valores que las instancias de tal atributo (contenidas por las instancias de la clase) pueden tener.

2.2.4.1.3 Operaciones Las operaciones son equivalentes a lo que introdujimos como métodos en los LPOO y son el elemento principal para describir el comportamiento de una clase.

2.2.4.2 Visibilidad

Los atributos y operaciones en UML poseen una visibilidad. La visibilidad especifica si el elemento puede ser accedido desde otras clases. En UML existen tres niveles de visibilidad: 1. pública: Cualquier clase con visibilidad a esta clase puede acceder al elemento con visibilidad pública. 2. protegida: Cualquier clase que descienda de esta clase la puede acceder. 3. privada: sólo el clasificador mismo puede utilizar el elemento.

2.2.4.3 Alcance

Otra característica importante de los atributos y operaciones es su propietario de alcance (en inglés: owner scope). El mismo indica si el atributo u operación es de clase o de instancia. De instancia significa que el elemento aparece u opera en cada instancia de clase, mientras si es de clase significa que existe un único elemento para todas las instancias de la clase. En el metamodelo de UML esto implica que el atributo `isStatic` sea `true`, lo que significaría de clase, o `false` lo que significaría que es de instancia.

2.2.4.4 Elementos abstractos, polimórficos y constantes

Se utilizan relaciones de herencia para modelar una jerarquía de clases, con clases más generalizadas en la parte superior de la jerarquía y otras más específicas en la parte inferior. Dentro de estas jerarquías, es común especificar que ciertas clases son abstractas, lo que significa que es posible que no tengan ninguna instancia directa. En el metamodelo de UML esto significa que el atributo `isAbstract` sea `true`. Una operación puede ser polimórfica, lo que significa que, en una jerarquía de clases, puede especificar operaciones con la misma firma en diferentes puntos de la jerarquía. Las implementaciones en las clases hijas anulan el comportamiento en las clases padre. Cuando se envía un mensaje en el tiempo de ejecución, la operación en la jerarquía que se invoca se elige de manera polimórfica; es decir, se determina una coincidencia en el tiempo de ejecución según el tipo de objeto. En UML, puede especificarse que una operación es abstracta (también con

el atributo `isAbstract`) lo que significa que no provee una implementación y la misma se posterga a clases hijas, o también puede ser polimórfica especificando una implementación, esto se consigue con el atributo `isLeaf` en `false`. Una operación con el atributo `isLeaf` en `true` significa que no es polimórfica.
Por último, una operación puede ser constante, esto significa que no modifica al objeto al que se le hace la llamada. Esto se consigue con el atributo `isConst` en `true`.

2.2.4.5 Paquetes

2.2.4.6 Modelando tipos primitivos

Los tipos modelados pueden ser traídos directamente del lenguaje de programación usado para implementar la solución, estos tipos son llamados tipos primitivos (en inglés primitive type).

2.2.4.7 Relaciones

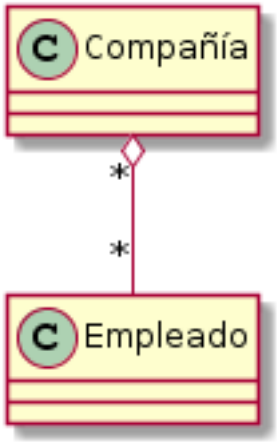
Al modelar un sistema ninguna clase queda aislada. Las mismas colaboran entre sí de varias formas. Por lo tanto, no sólo se deben identificar las cosas que forman el vocabulario del sistema (modelado en clases), por el contrario se deben identificar cómo estas clases se relacionan la una con la otra. UML define cuatro tipos de relaciones: Dependencias, asociaciones, generalizaciones y realizaciones.

2.2.4.7.1 Dependencia Una dependencia es una relación semántica entre dos cosas en donde un cambio en la semántica de uno (la cosa independiente) puede afectar la semántica de otra cosa (la cosa dependiente). La dependencia se modela con una flecha punteada. El siguiente diagrama muestra una clase B que depende de una clase A.

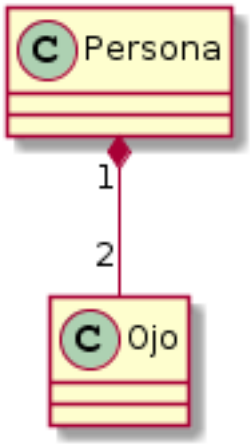
2.2.4.7.2 Asociación Una asociación es una relación estructural que describe un conjunto de vínculos entre dos clases. Una asociación contiene una multiplicidad (una instancia de la clase A con cuantas instancias de la clase B puede estar vinculada y viceversa). La siguiente figura nos muestra una asociación entre una clase A y otra B donde cada A se vincula con varios B (representado por el asterisco) y cada B con un solo A (representado por el número uno).

Las asociaciones pueden tener nombres en sus extremos que representan el nombre con el que se accede al tipo asociado. Los atributos cuyos tipos son clases pueden ser modelados mediante asociaciones.

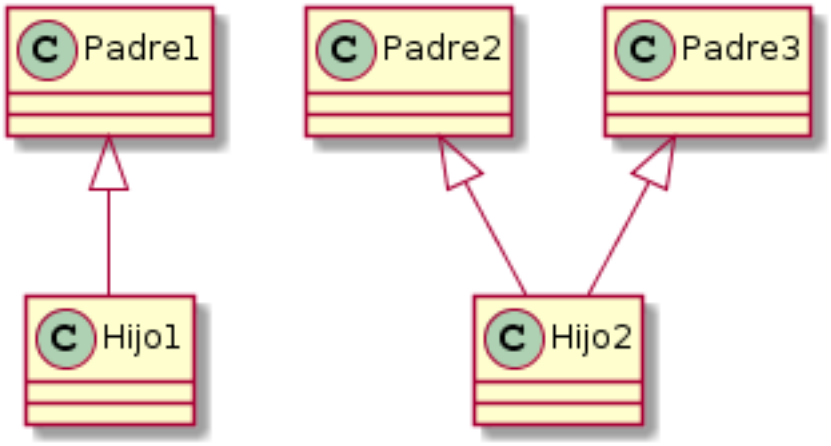
2.2.4.7.2.1 Agregación Una agregación es un tipo especial de asociación que representa una relación estructural entre el todo y sus partes. Una agregación adiciona la semántica "es un" a una asociación. La agregación es representada por un rombo vacío como en la siguiente figura.



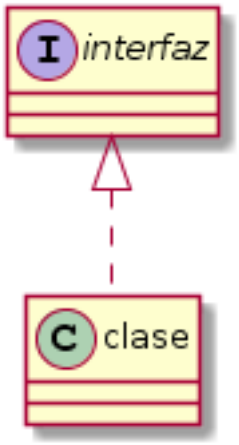
2.2.4.7.2.2 Composición Una composición es un tipo de agregación pero que agrega otra semántica a la asociación. Expresa quién es el poseedor de la parte y que el tiempo de vida de la parte está contenido dentro del tiempo de vida del todo (la parte puede ser instanciada junto con el todo o después y es destruida junto al todo o antes). La composición es representada con un rombo lleno como en la siguiente figura



2.2.4.7.3 Generalización Una generalización es una relación de generalización o especialización en la que los objetos del elemento especializado (el hijo o child en inglés) son sustituibles por los objetos del elemento generalizado (el padre o parent en inglés). De esta manera, la clase hija comparte la estructura y el comportamiento de la clase padre.
El siguiente diagrama nos muestra a la izquierda un ejemplo de herencia simple y a la derecha un ejemplo de herencia múltiple.



2.2.4.7.4 Realización Una realización es una relación semántica entre interfaces y clases, en donde una interfaz especifica un contrato que otra clase garantiza realizar. Una interfaz se modela igual que una clase pero con otro estereotipo (los estereotipos son explicados en la sección siguiente). La realización es modelada como una generalización con líneas punteadas, el siguiente diagrama nos muestra un ejemplo de ello.



2.2.4.8 Estereotipos

UML es un lenguaje de modelado general que busca poder describir la mayoría de los sistemas que uno debe modelar. A veces, es necesario introducir nuevas cosas que hacen parte del vocabulario del dominio del problema o de la solución o implementación y se ven como bloques primitivos de construcción (como las clases, paquetes o relaciones). Los estereotipos son aplicados a un elemento ya existente de UML cambiando o agregando significado a dicho elemento cuando se le asocia con el estereotipo. La siguiente figura nos muestra como se ve un elemento clase cuando se le agrega un estereotipo.

El estereotipo `metaclass` es aplicado al elemento clase y con el mismo puede describirse metaclases como `Metaclase` en el ejemplo.

2.2.4.8.1 Valores Etiquetados Al igual que los elementos de UML tienen asociados distintas propiedades, por ejemplo, las clases tienen asociadas un nombre. Así también los estereotipos pueden tener asociados

nuevas propiedades definidas para esos estereotipos.

2.2.4.9 Interfaces

Hay distintas formas de representar una interfaz en UML. La forma que utilizaremos en esta tesis es a través de un estereotipo aplicado a una clase.

2.2.4.10 Diagramas de clase

Un diagrama de clase muestra un conjunto de clases, interfaces, paquetes y sus relaciones. Los diagramas de clase son los diagramas más comunes en el modelado de sistemas orientados a objetos. Los diagramas de clase son utilizados para ilustrar la vista de diseño estático de un sistema.

Los diagramas de clase son importantes no solo para visualizar, especificar y documentar modelos estructurales, sino también para construir sistemas ejecutables mediante ingeniería directa e inversa. (Grady Booch 1998)

2.2.5 Ingeniería directa y diagramas de clase

UML no especifica un mapeo particular a ningún lenguaje de programación orientado a objetos, pero UML fue diseñado con tales mapeos en mente. Esto es especialmente cierto para los diagramas de clase, cuyos contenidos tienen una asignación clara a todos los lenguajes orientados a objetos de uso industrial, como Java, C ++, Smalltalk, Eiffel, Ada, Object Pascal y Forte (Grady Booch 1998).

La ingeniería directa es el proceso de transformación de un modelo en código a través de un mapeo a un lenguaje de implementación. Para generar código a partir de un diagrama de clase los autores de UML dan las siguientes directivas:
• Identifique las reglas para mapear su idioma de implementación con UML. Esto es algo que querrá hacer por su proyecto o su organización en general.
• Dependiendo de la semántica de los lenguajes que elija, es posible que deba

restringir el uso de ciertas funciones UML. Por ejemplo, UML le permite modelar herencia múltiple, pero JAVA permite sólo herencia simple. Puede elegir restringir a los desarrolladores el modelado de la herencia múltiple (lo que hace que sus modelos dependan del lenguaje) o desarrolle modismos que transformen estas características en el lenguaje de implementación (lo que hace que el mapeo sea más complejo).

- Utilice valores etiquetados para especificar su idioma de destino. Puede hacerlo a nivel de clases individuales si necesita un control preciso. También puede hacerlo en un nivel superior, como en los paquetes.
- Usa herramientas de ingeniería directa para los modelos.

En esta tesis buscaremos estudiar si es posible realizar ingeniería directa en C valiéndonos de frameworks o especificaciones para codificar C OO para que el mapeo entre UML y el código sea de forma intuitiva. Dependiendo el framework que utilicemos este mapeo puede ser más o menos complejo, pero minimizaremos su complejidad mapeando sólo a la semántica soportada por el framework.

Capítulo 3

Programación orientada a Objetos en C

3.1 El uso de C en el siglo XXI

A pesar de su antigüedad, C es ampliamente utilizado, tanto como lenguaje de iniciación a la programación como para producción de sistemas operativos, sistemas restringidos o con alta demanda de eficiencia y sistemas embebidos. La popularidad de este lenguaje puede medirse en su trayectoria en el índice TIOBE (index 2018) (al momento de la escritura de esta tesis en el segundo lugar) y aún más desproporcionada es su preponderancia en los sistemas embebidos: es el lenguaje de programación principal para aproximadamente el 70% de los diseñadores de firmware. Una revisión longitudinal de las encuestas de la industria que abarcan desde 2005 hasta 2018 muestra que C no sólo fue de manera holgada el idioma más utilizado, sino que en realidad aumentó su participación de mercado del 50% al 70% aproximadamente en esos años. Dentro de la comunidad de sistemas embebidos, parece que el año pico para C++ (segundo lenguaje utilizado) fue 2006 (Barr 2018).

3.2 La portabilidad y eficiencia del código escrito en lenguaje C

"C tiene las ventajas de una gran disponibilidad de compiladores para una gran variedad de procesadores y una bien merecida reputación de eficiencia en tiempo de ejecución."(Douglass 2010).

Más allá de estar trabajando para una arquitectura que disponga de compiladores o intérpretes para otros lenguajes, al elegir desarrollar en lenguaje C el código generado será portable a otras arquitecturas que podrían no disponer de otros compiladores o intérpretes. Claro que, si el código escrito hace referencia a la plataforma o entorno, ese mismo código no será portable a otras plataformas, pero el código que no lo hace puede ser compilado con cualquier compilador C para otra plataforma.
Otro punto en la potabilidad es que las librerías en C pueden ser ejecutadas por una gran variedad de otros lenguajes (mediante una interfaz de función exterior (FFI)).
Otra aclaración necesaria es que existen distintos estándares ISO de C (a saber ISO C89, ISO C99 e ISO C11), los más modernos pueden todavía no ser soportados por todas las plataformas mientras que el ISO C89 es altamente conocido por su disponibilidad y portabilidad (Deniau 2009) Por supuesto que este soporte del lenguaje por parte de tantas plataformas se debe a la simpleza y popularidad del mismo.
Siendo así, programar orientado a objetos en lenguaje C puede ser más conveniente que hacerlo bajo otros lenguajes aunque nativamente soporten este paradigma.

3.3 Programación orientada a objetos en C

Bajo el paradigma de orientación a objetos se han creado distintos lenguajes como ser C++, Java, SmallTalk, CLOS y muchos más. En cambio el lenguaje C fue concebido bajo el paradigma de programación estructurada.
La programación estructurada no contiene ninguno de los conceptos de la

programación orientada a objetos que hemos visto. Sin embargo, la siguiente pregunta surge: ¿puede programarse orientado a objetos con un lenguaje estructurado? Douglass compara esto a cuando antiguamente se hacían la misma pregunta con respecto al lenguaje ensamblador, ¿podría ser utilizado para programar en forma estructurada? La respuesta a ambas preguntas es sí, tan sólo se debe buscar la forma de implementar los conceptos de cada paradigma en el lenguaje deseado (Douglass 2010).

La orientación a objetos es más que una forma de programar, es una forma de pensar, es un paradigma. Cuando la POO es vista como un enfoque, una forma de pensar, entonces estamos hablando del análisis y diseño del software. De esta manera la POO se convierte principalmente en lo que es conocido como Análisis y Diseño orientado a objetos (OOA&D por sus siglas en inglés). (Hendrickx 2004)

Cada uno de estos son parte de los diferentes aspectos que posee la OO: análisis, diseño, programación, etc. Los distintos aspectos de la OO pueden ser independientes entre sí. Esto significa:
1. Un diseño puede ser Orientado a Objetos, incluso si el programa resultante no lo es (Madsen 1988).
2. Un programa puede ser Orientado a Objetos, incluso si el lenguaje en el que está escrito no lo es (Madsen 1988).
3. Un programa Orientado a Objetos puede ser escrito en casi cualquier lenguaje, pero un lenguaje no puede ser asociado con la orientación a objetos amenos que promueva programas Orientados a Objetos (Stroustrup 1991).

En esta tesis la Programación Orientada a Objetos (POO) no significa usar un Lenguaje de Programación orientado a objetos (LPOO), sino que significa implementar un Diseño Orientado a Objetos en el lenguaje de programación C, lo que resulta en un programa Orientado a Objetos(OO).

Los LPOO proveen mecanismos para soportar los conceptos de la POO y un estilo de programación en consonancia con los mismos. Tales LPOO simplifican el trabajo de implementar un diseño OO, pero un LPOO no es obligatorio para tales diseños. Implementar un diseño OO puede ser hecho en prácticamente cualquier lenguaje. Tal lenguaje puede ser C (Sommerville 1996), pero no significa que C sea un LPOO, tan solo permite

al programador escribir programas OO (Stroustrup 1991). La opinión de Bjarne Stroustrup es la más popular afirmando que un lenguaje que permita escribir programas OO no debería ser considerado para tal función si se requiere de un esfuerzo excepcional para hacerlo (Stroustrup 1991). Esta opinión es un tanto injusta, ya que un lenguaje es solo una herramienta en el conjunto de las que pueden utilizarse para crear programas OO. Las dificultades que presenta el lenguaje para expresarse en una manera OO puede ser suplida por una o varias herramientas (Hendrickx 2004).

La OO en C es ampliamente utilizada hoy día, Bruce Powel Douglass escribió un libro de Patrones de Diseño que utilizan la orientación a objetos para sistemas embebidos en C (Douglass 2010).

3.4 Dificultades en la programación orientada a objetos en lenguaje C

La codificación disciplinada de C orientada a objetos es muy propensa a errores. Por ejemplo, la forma más utilizada de conseguir polimorfismo es a través de punteros a función (la especialización puede ser fácilmente hecha con declaraciones switch case pero es mucho más versátil si, en cambio, se utilizan punteros a función) (Douglass 2010). "La desventaja es que los punteros a función son engañosos y los punteros son la causa principal de los defectos introducidos por el programador en programas en C" (Douglass 2010). Cada función debe ser referenciada por el puntero correcto y a veces se debe inicializar los punteros en una lista, por lo que un cambio de orden puede ser fatal. También, la cantidad de código que se debe escribir para que una clase en C contenga estos conceptos es significativa (A.-T. Schreiner 1993). Si bien esto ha llevado a mal juzgar la capacidad de C como un lenguaje alternativo para el paradigma de orientación a objetos, no se debe juzgar al lenguaje en aislamiento sino a todo el toolsuite de desarrollo (Hendrickx 2004).
En el siguiente capítulo veremos en detalle las dificultades que se presentan para codificar COO de acuerdo a distintos frameworks, luego presentaremos

un generador de código desde diagramas de clase UML como facilitador para dicha codificación.

- Debido a la preponderancia de C en los sistemas embebidos buscamos mostrar el interés especial que tienen herramientas de este tipo para los mismos.*

3.5 La necesidad de diseño en los sistemas embebidos

"Actualmente, la creciente complejidad del proceso de desarrollo de software embebido, demanda técnicas de diseño capaces de manejar tal complejidad eficientemente" (Lennis & Aedo 2013) Esta eficiencia es crítica para reducir el costo de desarrollo ya que "cerca del 80% del costo en el desarrollo de sistemas embebidos es atribuido a cuestiones de diseño"(ITRS 2011).
Debemos preguntarnos cuál es el motivo de tales esfuerzos en el diseño. Para eso debemos analizar las características intrínsecas de los sistemas embebidos que hacen del diseño algo esencial.
"Una de las más notables características de los sistemas embebidos es la enormidad de sus restricciones.[...] recursos como la memoria, alimentación, refrigeración, o poder computacional contribuyen al costo de cada unidad" (Douglass 2010). "Las presiones para agregar capacidades [al sistema] mientras simultáneamente reducir los costos implica que los desarrolladores de embebidos deben continuamente buscar maneras de mejorar sus diseños, y su habilidad de diseñar eficientemente"(Douglass 2010). Si deseamos realizar cambios, extendiendo el sistema o bajando su consumo de recursos, lo recomendable es tener un diseño con bajo impacto a los cambios y al mismo tiempo eficiente. También, distintos artefactos de software deben correr en distintas plataformas. Esto lo hemos visto "con la caída de los costos de hardware, sistemas originalmente hechos en pequeños procesadores de 8 bit se han actualizado a procesadores de 16 ó 32 bit más capaces. Esto es necesario para agregar la complejidad adicional de comportamiento requerido en los dispositivos embebidos modernos y se hace posible por los bajos costos recurrentes de partes y manufactura" (Douglass 2010).
Un tercer punto a tener en cuenta es que "muchos proyectos de sistemas

embebidos involucran el desarrollo de hardware electrónico y mecánico simultáneamente con el software. Esto introduce un desafío especial para el desarrollador que debe elaborar el software solamente sobre la base de especificaciones de prediseño, de cómo el hardware debería trabajar. Muy frecuentemente, esas especificaciones son nada más que nociones de funcionalidad, haciendo imposible crear el software correcto hasta que el hardware es entregado." (Douglass 2010). Estos problemas pueden ser atacados eficientemente con un diseño orientado a objetos. Una inversión en la cadena de dependencias posibilita que la implementación del negocio a resolver sea independiente del hardware. A la vez, identificar por dónde vendrán los cambios en las capacidades del sistema o sus optimizaciones, junto con un buen diseño orientado a objetos permite que el impacto en el código original por los cambios sea minimizado, reduciendo los costos de dichos cambios.

Además de esto, para tener un testing efectivo se debe pensar en un diseño orientado a las pruebas. Bruce Powel Douglass, gurú de diseño y procesos para sistemas embebidos, recomienda utilizar TDD (desarrollo dirigido por las pruebas) (Douglass 2010). James W. Grenning es autor de "Test Driven Development for Embedded C", un libro dedicado a este asunto para sistemas embebidos. Él nos escribe: "El componente central de TDD son las pruebas unitarias, sin un diseño modular que permita descomponer el sistemas en partes no sería esto posible" (Grenning 2011). La modularidad es un elemento básico en la orientación a objetos contenida en el concepto de encapsulamiento.

Toda propuesta de cómo implementar características de orientación a objetos en C embebido debe seguir la normativa de no hacerlo a costa de un obligado incremento en el costo de hardware. Al mismo tiempo, para sistemas complejos y menos restringidos se debería permitir contar con más facilidades para el desarrollador para que haga frente a tal complejidad con un menor costo de diseño, de ser necesario cediendo algo de eficiencia.

Cualquier avance en los procesos de desarrollo de software embebido en C implicará un importante impacto en la mayoría de los desarrollos en sistemas embebidos actuales.

3.6 Definiciones en COO

A continuación presentaremos algunas definiciones para formar un lenguaje en común que nos permita describir las distintos frameworks de COO de forma más sencilla.

3.6.1 Herencia de estructuras

Ya definimos de que se trata la herencia de clases en la POO. El mismo concepto podemos aplicarlo a las estructuras en C y es la base en todas las implementaciones para la herencia de clases en COO. Uno puede declarar una estructura A y luego definir funciones que reciban en sus argumentos a dicha estructura. Si esta estructura es incluida dentro de otra estructura B como primer variable de la estructura, entonces todas las funciones escritas para la estructura A servirán también para la estructura B. Es cierto que el compilador de C puede advertirnos de estar usando una estructura B en vez de una estructura A, pero si los argumentos son referencias un simple casteo de una referencia de B a una de A puede solucionarlo.

```
struct A{
    char * name;
};
struct B{
    struct A a;
    int age;
};
void doSomethig(struct A * a);
/*...*/
struct B b;
doSomethig((struct A*) &b);
```

La contra de esta técnica es que para acceder a un dato en A desde una estructura B hay que utilizar un nombre más de referencia. Por ejemplo para acceder a la variable `name` en `b` se debe escribir `b.a.name`, y si seguimos

agregando más capas de herencia esto se vuelve muy molesto para el programador. Hay distintas maneras de mitigar este problema, por ejemplo codificando macros o funciones que manipulen dichos datos a partir de referencias a las estructuras.
En el capítulo siguiente veremos otras dos formas de resolver esto una utilizando estructuras anónimas de ISO C11 y la otra utilizando macros de listas de los miembros de las estructuras.

3.6.2 Estructura de instancia de clase

Instanciar una clase significa poder tener una representación en memoria de un posible estado de una clase, o sea que sus atributos de instancia tengan algún valor permitido por la clase. Si deseamos poder realizar esto en lenguaje C deberemos tener una estructura que contenga todos los atributos de la clase y que instanciar dicha estructura sea uno de los requisitos para obtener una instancia de dicha clase (luego para finalizar la instanciación se deberá llamar a una función que haga de constructor de la misma). A dicha estructura la llamaremos **estructura de instancia de clase**.
Esta estructura puede no tener el mismo nombre de la clase y es posible que la estructura de RTTI (ver a continuación) sea la que tiene dicho nombre.

3.6.3 Tabla virtual

Como ya vimos, el concepto de polimorfismo implica que bajo el mismo mensaje enviado a un objeto o, lo que es lo mismo, instancia de clase, se ejecuta una función dependiente del tipo de clase de la cual pertenece el objeto. La manera más utilizada para implementar esto es que junto con las variables de instancia en la estructura de instancia de clase, haya una referencia que permita referenciar a una tabla de punteros a función. Cada clase, entonces, proveerá su propia tabla y en el lugar donde una asigna cierta función para ser llamada ante cierto mensaje, otra podrá asignar otra función. Dicha tabla es llamada **tabla virtual**.
En el siguiente capítulo veremos distintas implementaciones de tablas virtuales que poseen distintas características.

Además introduciremos 2 frameworks (Dynace y COS) que utilizan otro enfoque para el soporte del polimorfismo, la de las funciones genéricas que introducimos en el capítulo anterior. Bajo este enfoque, la función encargada de recibir el mensaje al objeto (el despachador) es la que contiene las referencias a función a ser ejecutadas para cada tipo de clase. Esto se consigue registrando cada función de implementación en el dispatcher.

3.6.4 Despachador o Selector

En general, existe una función que representa a cada mensaje que puede ser aplicado a todos o algunos objetos. La misma es encargada de ejecutar la función correspondiente para el tipo de clase del objeto. Esta función es llamada **despachador** o **selector** (o en inglés **dispatcher** o **selector**).

3.6.5 Estructura de RTTI

Es común que en los lenguajes orientados a objetos exista una referencia a la clase misma a la cual se la puede interrogar para obtener su nombre y miembros y estructura. En C esta referencia será a la instancia de una estructura que pueda contener dicha información. Llamaremos a dicha estructura **estructura de RTTI**.

Capítulo 4

Estado del arte en programación orientada a objetos en C

4.1 Variedad de frameworks y especificaciones

Un LPOO conoce conceptos del paradigma de OO y provee mecanismos para facilitar el desarrollo en base a esos conceptos. A pesar de que C no conoce los conceptos del paradigma de OO de por sí, de todas formas es posible facilitar el desarrollo con sus conceptos.(Stroustrup 1991) Esto se debe a dos factores. El primero es que C es un lenguaje de programación muy flexible. Y, en segundo lugar, los modelos de objetos que permiten soportar los conceptos de orientación a objetos son perfectamente implementables en C. En esta sección veremos como se ha especificado la codificación en C bajo ese paradigma por distintos autores, los distintos conceptos que han podido ser implementados por cada uno, el modelo de objetos que los contiene.

Por último remarcaremos qué dificultades presenta cada una de estas especificaciones para el programador C. Y propondremos una representación en UML para dicha codificación libre de esas dificultades. Las especificaciones que analizaremos son:

1. La de Ben Klemens en su libro 21st Century C (Klemens 2013)
2. SOOPC de Miro Samek (Samek 2015)
3. OOPC de Laurent Deniau (Deniau 2001)
4. OOC de Tibor Miseta (Miseta 2017)
5. OOC-S de Laurent Deniau (Deniau 2007b)
6. OOC de Axel Tobias Schreiner (A. T. Schreiner 1993)
7. GObject de glib (GLib 2019)
8. Dynace de Blake McBride (McBride 2004)
9. COS de Laurent Deniau (Deniau 2009).

Para cada especificación enumeraremos los conceptos de orientación a objetos soportado. Esto es importante, como dijimos en el resumen, para analizar a C como un lenguaje apropiado para la enseñanza de la orientación a objetos. Un único lenguaje, simple en sus reglas y muchas veces el primero introducido a los estudiantes, que bajo algunos de estos frameworks se incorporan los conceptos para su enseñanza. Al ser frameworks de código abierto, los mecanismos por los cuales se logra tal soporte de conceptos quedan al descubierto, mostrando así los costos de dicho soporte, en forma quizás aproximada, para cualquier otro LPOO.
Quien desee obtener una mejor introducción del framework para poder codificar bajo el mismo, se recomienda buscar su documentación en las referencias dadas para el mismo.

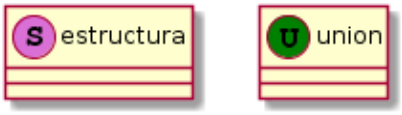
4.2 Modelos de objetos

Los modelos de objetos nos muestran como están organizados los datos que dan soporte a la implementación de los conceptos de orientación a objetos de cada framework.
Para los frameworks más adecuados para sistemas altamente restringidos hemos incluido un análisis detallado de las estructuras que dan soporte al framework. Esto permite evaluar al framework en su uso de memoria (RAM o ROM) y uso de tiempo de CPU (que depende de la cantidad de indirecciones existentes para llegar a los datos). Se han representado, con explicaciones, en diagramas de clase UML.

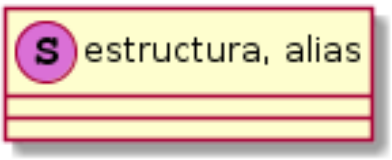
4.2.1 Cómo leer los modelos de objetos

A la hora de elegir los nombres para las estructuras, variables y referencias, para facilitar la introducción al framework, se eligieron los mismos que se utilizan en su codificación en C.
Algunas diferencias con un diagrama de clases UML de acuerdo al estándar son:

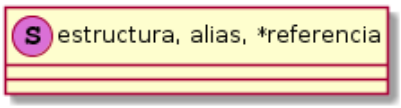
- Cada estructura o unión en C es representada por una clase en el diagrama. Las uniones con una U y las estructuras con una S.



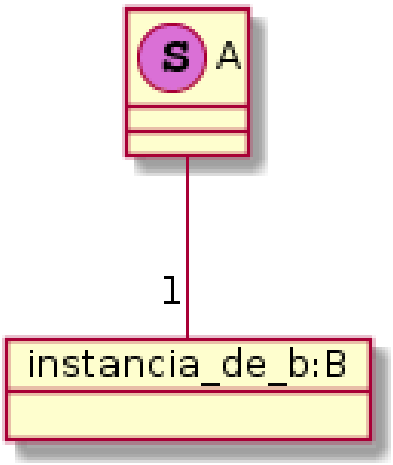
- El nombre de una estructura en C puede tener uno o varios alias, que son representados en el diagrama UML por una lista separada por comas en el nombre de la clase.



- El alias de una estructura en C puede ser un nombre que se utiliza para representar un puntero a la misma, representado en el diagrama por un * seguido por el nombre del alias.



- Cuando todas las instancias de una estructura A (no heredada) referencian a la misma instancia de otra estructura B, esto se representará asociando la clase que representa a la estructura A con una instancia de B (objeto UML).



- Los arreglos se identifican bajo el estereotipo `<<array>>`. Un arreglo posee un tipo. Los arreglos pueden componerse de instancias de su tipo y definen los elementos del arreglo.

Representaremos la herencia de estructuras como una composición de estructuras indicando que se trata del primer elemento de la estructura.

4.3 Codificación

Para cada framework se mostrarán ejemplos de codificación bajo los mismos para facilitar el análisis. Para ellos hemos creado pequeños prototipos ejecutables que pueden encontrarse en el repositorio especificado en el apéndice.

4.4 Dificultades en la codificación

El estudio de estos frameworks busca explorar si desde el modelo de objetos más sencillo (que soporta menor cantidad de conceptos de orientación a objetos) hasta el más complejo (que soporta una mayor cantidad de conceptos), la dificultad de implementar los conceptos que define (clases, interfaces, etc.) es grande, incluso haciendo uso intensivo del preprocesador de C. Presentaremos una representación en UML para la codificación para cada framework que no presenta dicha dificultad.
Dentro de cada análisis incluiremos las "dificultades en la codificación" que son las dificultades para prototipar una clase de acuerdo a su relación con otras clases o interfaces, declarar sus métodos sin cuerpos y modificar cualquiera de estas cosas.
Una dificultad que nombraremos aquí por ser común a todos los frameworks al estar basados en C es que cualquier inicialización o destrucción de atributos con tipos de clase debe ser codificada manualmente ante la inexistencia de un mecanismo automático que lo realice. Además la utilización dentro de un método de un objeto alocado en memoria automática también precisa de una inicialización y destrucción explícitas.

4.5 Propuestas de expresión en UML

Para cada framework se dará una propuesta de expresión bajo UML. Con la misma se busca mostrar que UML no posee las dificultades estudiadas. Además nos servirá como base para el estudio de generación de código desde UML para estos frameworks. Las propuestas de expresión en UML estarán basadas en la definición de UML 2.5.1 dadas por la OMG (OMG 2017).

4.6 Ben Klemens

4.6.1 Introducción

En su libro (Klemens 2013), busca mostrar nuevas formas de resolver problemas en lenguaje C de acuerdo a los nuevos estándares C99 y C11, y con las bibliotecas actuales de desarrollo en C como glib. Busca conseguir una sintaxis legible y mantenible para su codificación, sin intentar imitar a otros lenguajes a costa de un preprocesamiento extensivo con el preprocesador de C. De acuerdo a su especificación, el usuario de las clases, obtiene un uso muy familiar con respecto a otros LPOO, llamando a los métodos polimórficos en la forma C++ o JAVA, aunque enviando el objeto como primer parámetro:

```
object->method(object);
```

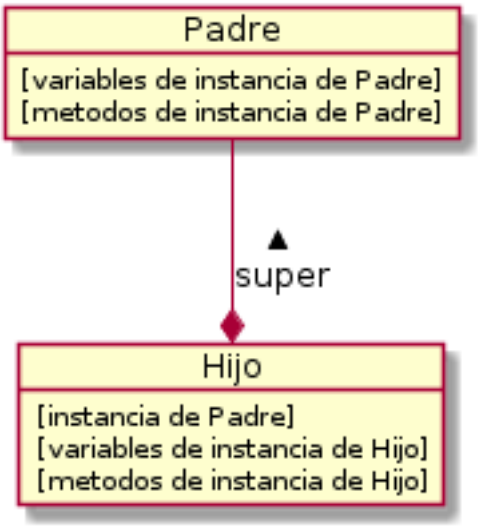
Aunque veremos como eliminar esta repetición mediante una macro.

4.6.2 Conceptos soportados

Los conceptos de la orientación a objetos soportados son mínimos:
1. Encapsulamiento
2. Herencia
3. Polimorfismo

4.6.3 Modelo de objetos

El siguiente diagrama es una representación del modelo de objetos. Es el modelo más simple en el que los datos (atributos y métodos) de instancia se encuentran en la misma estructura y no hay referencias nativas a datos de clase (aunque veremos como representarlos).



La herencia representada en la figura como la relación Padre-Hijo se consigue incluyendo una instancia del Padre en el comienzo de la instancia de Hijo. De esta manera las funciones y métodos escritos para las instancias de Padre se pueden utilizar para las instancias de Hijo.
No existe una estructura que contenga datos de clase, aunque sí se podría referenciar a estos con punteros. Por ejemplo, si se desea definir una variable de clase entera `i` todos los objetos podrían contener un puntero al mismo llamado -por ejemplo- `class_i`. Lo mismo es aplicable a métodos de clase. Comparado con un modelo de objetos basados en tabla virtual como los siguientes 5, o como otros LOOP como C++ o JAVA, este modelo generalmente presentará un mayor consumo de memoria RAM ya que todas las referencias a métodos son copiadas individualmente en la estructura de la instancia. Por otro lado, ejecutar estos métodos polimórficos consumirá menos tiempo ya que no es necesario desreferenciar la tabla virtual. Además, las referencias individuales se pueden utilizar para cambiar los métodos a ejecutar en tiempo de ejecución en base al estado del objeto, lo que puede describir más limpiamente el comportamiento dependiente de estados como sucede con lenguajes prototipados como JavaScript.

4.6.4 Codificación

La declaración de una clase base, es decir que no hereda de ninguna otra, no es más que la declaración de una estructura en C con sus métodos representados en punteros a función y junto a 3 métodos de instanciación (new), copia (copy) y liberación (free) de objetos.
A continuación un ejemplo con una clase Parent

```
//parent.h
typedef void (*method_p_t)(parent*self);
typedef struct parent{
    int private_attribute1;
    int attribute2;
    method_p_t method;
}parent;

parent * parent_new();
parent * parent_copy(parent * parentToCopy);
void parent_free(parent * parentToFree);
```

Para representar la visibilidad de los miembros no públicos de la clase Klemens recomienda no usar ningún artilugio (como veremos de otros frameworks más adelante) y poner la confianza del lado del usuario de la clase indicando en el nombre del miembro su visibilidad (scope), así el atributo privado `attribute1` se llamará `private_attribute1`.

La implementación de la clase consiste en la implementación de los métodos new, copy, free; la implementación de los métodos polimórficos y su asignación al puntero a función que los representa

```
//parent.c
static void method(parent * self){
    //do something
}
parent * parent_new(){
```

```
    parent * out = malloc(sizeof(parent));
    *out = (parent){ .method = method, .private_attribute1 = 0,
            .attribute2 = 0};
    return out;
}

parent * parent_copy(parent * in){
    parent *out = malloc(sizeof(parent));
    *out = *in;
    return out;
}
void parent_free(parent * in){
    free(in);
}
```

Más métodos no polimórficos pueden definirse para la clase en la forma de new, copy o free.
El uso de la clase es muy intuitivo para usuarios de otros LPOO:

```
parent * parent1 = parent_new( );
parent1 -> attribute2 = 10;
parent * parent2 = parent_copy( parent1 );
parent2 -> method ( parent2 );
parent_free ( parent1 );
parent_free ( parent2 );
```

El siguiente código nos muestra como definir una clase que herede de otra.

```
//child.h
typedef struct child{
    union{
        struct parent;
        struct parent asParent;
    };
```

```
    //new attributes and polymorphic methods...
}child;
//new(),copy(),free() and other non polymorphic methods...
```

Las uniones y estructuras anónimas nos permiten acceder directamente a los miembros de la misma sin necesidad de referenciarlas a través de otros nombres. Así para llamar a la función polimórfica `method()` definida en `parent` a través de una instancia de `child` llamada `aChild` va a tener la misma forma que para una instancia de `parent`: `aChild -> method ( aChild )`, aunque este generará un conflicto de tipo ya que el método `method()` espera un primer argumento de tipo `parent` y no de tipo `child`. Se incluye un acceso referenciado por nombre (`asParent`)a la estructura padre (`parent`) para poder resolver los conflictos de tipo para los métodos definidos en clases padres. De esta manera llamando a `aChild -> method ( aChild.asParent )` eliminaremos el conflicto de tipo que generaba el ejemplo anterior.
Aunque Klemens no lo menciona, con esta especificación podemos generar macros que realicen verificaciones de tipo en tiempo de compilación para enviar mensajes a los objetos:

```
//parent.h
#define _method(_parent) _Generic((_parent),                    \
                parent:_parent -> method ( _parent ),           \
               default:_parent -> method ( _parent.asParent ) )
```

`_Generic` es una nueva facilidad de C11 que permite evaluar una variable de acuerdo a su tipo. Con esta macro, para enviar el mensaje `method()` a una instancia de parent o a un objeto que herede de parent (por ejemplo `aChild`), no se necesita más que llamar a `_method(aChild)`. Si el argumento que pasamos no es un `parent` o una clase heredada del mismo entonces obtendremos un error en tiempo de compilación por no encontrarse el miembro `asParent` en su estructura.

La implementación de los métodos `new()` y `free()` deben llamar a los métodos `new()` y `free()` de la clase padre, para `new()` al principio del método y para `free()` al final:

```c
//child.c
static void method(parent * self){
    //do something
}
child * child_new(){
    child * out = (child *) parent_new();
    out -> method = method;
    // other initialization for child
    return out;
}

void child_free(child * in){
    // child release of resources
    parent_free(&in->asParent);
}
```

En el código de ejemplo vemos como `child` puede asignar su propia implementación del método polimórfico `method()` por lo que `_method(aParent);` llama a la función `method()` definida en parent.c mientras que `_method(aChild)` llama a la función `method()` definida en child.c.

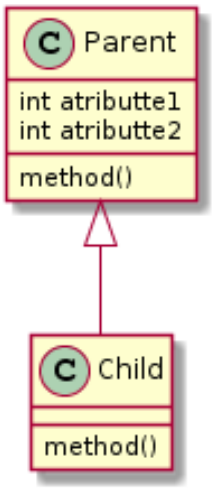
4.6.5 Dificultades en la codificación

#	Tipo de dificultad	Nombre	Descripción
1	Propensión a errores	Inicialización de métodos polimórficos	Contrario a otros LPOO a la definición de un método debe asignársele su implementación en el método constructor de la clase (`new()`), si esto se olvida generaremos un error crítico al enviar un mensaje con el método no inicializado.

#	Tipo de dificultad	Nombre	Descripción
2	Propensión a errores y obtención de información	Prototipado de métodos polimórficos redefinidos	Para crear el prototipo de un método ya definido por una clase padre se debe conocer exactamente cuál de los padres es el que definió el método por primera vez, además de que en caso de equivocarnos recibiremos una advertencia por parte del compilador, esto agrega una demora al programador que debe obtener esta información.
3	Repetición de código	Cada método implica de tres a cinco referencias al mismo	Incluirlo en la estructura de la clase, escribir su implementación, asignar la implementación al puntero a función en la estructura que lo representa y si es la primer clase que define el método, implementar su dispatcher.
4	Repetición de código	Nombre de la clase en cada método no polimórfico	Para no colisionar con los nombres utilizados por otras clases, todas las funciones llevan por delante el nombre de la clase
5	Repetición de código	Repetición en los métodos new, copy, free	Los métodos `new`, `copy` y `free` de cada clase son prácticamente iguales. Esto puede solucionarse con una macro.
6	Repetición de código	Heredar de una clase implica mucho código	Comparado con otros LPOO donde no hace falta más que una palabra reservada, el nombre de la clase padre e hija, esta especificación requiere el uso de nueve palabras. Esto puede solucionarse con una macro.

4.6.6 Propuesta de expresión en UML

El diagrama UML que puede contener estos conceptos es directa, ya que son los más básicos para la orientación a objetos.
El siguiente diagrama los expresa de forma correcta.



Para especificar que el método es polimórfico el atributo `isLeaf` debe ser falso de acuerdo a la definición de UML (OMG 2017) . La herencia se indica como de costumbre en UML. Para especificar en el diagrama que Child contiene su propia implementación del método `method` se lo vuelve a incluir como miembro de Child. Se puede indicar la visibilidad de los miembros lo que implica una indicación en su nombre.
Ninguna de las dificultades enumeradas en la sección anterior aplican para este diagrama.

4.7 Simple Object-Oriented Programming in C (SOOPC)

4.7.1 Introducción

OOPC fue ideado por el autor como una forma simple de representar los conceptos centrales de la POO en APIs escritas en C. Busca ocultar la

dificultad de implementar POO en C para el desarrollador de la aplicación. Es compatible con el estándar ISO C89.

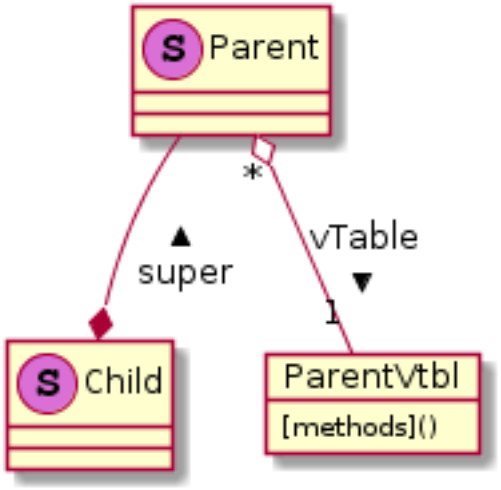
4.7.2 Conceptos soportados

Los conceptos de la orientación a objetos soportados son mínimos:

1. Encapsulamiento
2. Herencia
3. Polimorfismo

4.7.3 Modelo de objetos

Así como la cantidad de conceptos soportados es mínima, también lo es el modelo de objetos necesario para soportarlos. La siguiente figura representa este modelo de objetos.



En el diagrama, `Parent` es una clase definida por el usuario, podría tener cualquier nombre. Debe contener una referencia a una tabla virtual si es que se trata de una clase con métodos polimórficos o variables de clase. `Child` es otra clase definida por el usuario que hereda de la clase `Parent`. Los datos de `Parent` deben ser los primeros en memoria de `Child`, esto permite que las funciones que reciben una referencia a `Parent` como argumento, se les pueda pasar un una referencia a `Child`. `Child`, a través de la referencia a la tabla

virtual heredada de `Parent`, referencia a su propia tabla virtual del mismo tipo que la tabla virtual de `Parent`. Si bien el autor de esta especificación no lo contempla, si la clase `Child` agregara más métodos virtuales para ella y para sus clases heredadas la referencia a la tabla virtual de `Child` podría ser un tipo heredado de `ParentVtbl`, una `ChildVtbl`, al igual que `Child` hereda de `Parent`.

4.7.4 Codificación

El modelo de objetos quedará más claro con los siguientes ejemplos de codificación de una clase y una clase heredada de esta. La codificación sigue la especificación del autor.

A continuación codificamos la tabla virtual (vtable) de una clase que no herede de ninguna otra. En la estructura agregamos los métodos polimórficos de instancia así como las variables de clase, el método `method` es un ejemplo.

```
/*Parent.h*/
typedef struct {
    void (*method)(const Parent * self);
    /*más métodos o variables de clase aquí*/
} ParentVtbl;
```

La clase Parent es representada a través de un la estructura Parent.

```
/*Parent.h*/
typedef struct {
    const ParentVtbl * vptr;
    /*variables de instancia van aquí*/
} Parent;
```

Si Parent no tuviese métodos polimórficos la referencia a una tabla virtual podría postergarse hasta que alguna clase heredada sí los tenga.

La inicialización de la referencia a la tabla virtual así como la tabla virtual misma se realiza en el constructor.

```
/*Parent.c*/
void Parent_metodo_(Parent * self)
{
    /*cuerpo del método*/
}

void Parent_ctor(Parent * self)
{
    static const ParentVtbl vtbl=
    {
        Parent_metodo_
    };
    self -> vptr = &vtbl;
}
```

Las implementaciones de los métodos terminan con guión bajo(_), esto es así para diferenciarlo de las funciones que llaman a dichos métodos (dispatchers).

Los dispatchers hacen las llamadas a los métodos referenciados por las tablas virtuales, y son los utilizados por los usuarios de las clases. Hay tres alternativas para implementarlos: como una función, como una función "inline" (en caso de trabajar con C99) o como una macro. A continuación la presentaremos como función:

```
/*Parent.c*/
void Parent_metodo(Parent * self)
{
    self -> vptr -> metodo (self);
}
```

Para implementar una herencia de la clase Parent se la ubica como primer miembro de la clase hija.

```
/*Child.h*/
typedef struct {
```

```
    Parent super;
    /* más variables de instancia van aquí*/
} Child;
```

El constructor realiza la misma tarea que el constructor de Parent:

```
/*Child.c*/
void Child_metodo_(Parent * self)
{
    /*cuerpo del método*/
}

void Child_ctor(Child * self)
{
    static const ParentVtbl vtbl=
    {
        Child_metodo_
    };
    Parent_ctor(&self->super);
    self -> vptr = &vtbl;
}
```

Vemos que primeramente se realiza una llamada al constructor de la clase padre y luego se inicializa la clase heredada al igual que en cualquier lenguaje orientado a objetos. En caso de precisar un destructor, el orden debe ser el inverso, primero se liberan los recursos de la clase hija y luego los de la clase padre. Vemos también que la referencia a la tabla virtual de Parent que se asigna en el constructor de Parent es luego reasignada con la tabla virtual de Child.

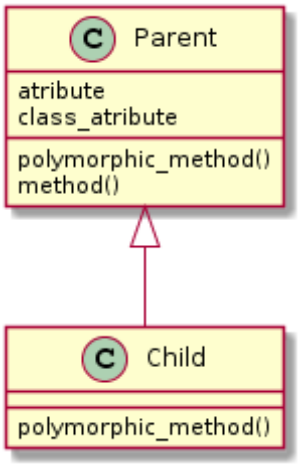
4.7.5 Dificultades en la codificación

#	Tipo de dificultad	Nombre	Descripción
1	Propensión a errores y obtención de información	Inicialización de la tabla virtual	Al ser una especificación compatible con C89 la única manera de instanciar la tabla virtual como constante obliga a inicializar sus miembros con una lista ordenada de valores, si el orden es incorrecto se estarán llamando a implementaciones de métodos que no corresponden a la llamada hecha, si el prototipo de los métodos es equivalente entonces no tendremos ningún aviso del compilador al respecto. Además se debe conocer la jerarquía de herencias y los métodos virtuales definidos por cada clase, esto agrega una demora al programador que debe obtener esta información.
2	Propensión a errores y obtención de información	Prototipado de métodos polimórficos redefinidos	Para crear el prototipo de un método ya definido por una clase padre se debe conocer exactamente cuál de los padres es el que definió el método por primera vez, además de que en caso de equivocarnos recibiremos una advertencia por parte del compilador, esto agrega una demora al programador que debe obtener esta información.

#	Tipo de dificultad	Nombre	Descripción
3	Repetición de código	código repetitivo para todas las clases	Instanciar una tabla virtual, asignarla a la referencia en el objeto son operaciones que se realizan para todas las clases y se deben repetir manualmente para el implementador.
4	Repetición de código	Cada método implica de 3 a 5 referencias al mismo	Incluirlo en los miembros de la tabla virtual, generar su implementación (puede precisar tener su prototipo en el archivo de encabezado (`.h`) si lo precisa una clase heredada), asignar la implementación a la tabla virtual y si es la primer clase que define el método, implementar su dispatcher.
5	Repetición de código	Nombre de la clase en cada método miembro	Para no colisionar con los nombres utilizados por otras clases, todas las funciones llevan por delante el nombre de la clase.

4.7.6 Propuesta de expresión en UML

El diagrama UML que puede contener estos conceptos es directa ya que son los más básicos para la orientación a objetos. El siguiente diagrama los expresa de forma correcta.



Para especificar que el método `polymorphic_method` es polimórfico la propiedad `isLeaf` debe ser falsa de acuerdo a la definición de UML (OMG 2017). La herencia se indica como de costumbre en UML. Para especificar en el diagrama que Child contiene su propia implementación del método `polymorphic_method` se lo vuelve a incluir como miembro de Child. Para que el atributo `class_attribute` sea de clase (o sea esté referenciado a través de la tabla virtual por todas las instancias de la clase), la propiedad `isStatic` debe ser verdadera de acuerdo a la definición de UML (OMG 2017). Cualquier especificación de visibilidad en los miembros de la clase o instancia son ignorados exceptuando los métodos no polimórficos que pueden ser públicos (incluidos en el archivo header de la clase) o privados (no incluidos en el archivo header de la clase). Ninguna de las dificultades enumeradas en la sección anterior aplican para este diagrama.

4.8 OOPC de Laurent Deniau

4.8.1 Introducción

Este framework está fuertemente basado en C++, por lo que su modelo de objetos y performance son muy parecidos a los de este LPOO. Fue ideado como alternativa a C++ bajo el estándar ISO C89 (en el caso donde no hay un compilador C++ disponible, como sucede para varios sistemas embebidos, o no se quiera utilizar el compilador C++ disponible como

tecnología de desarrollo o para no depender de la API de C++), aunque también presenta facilidades adicionales bajo el estándar ISO C99, y además provee interesantes facilidades para depurar.

4.8.2 Conceptos soportados

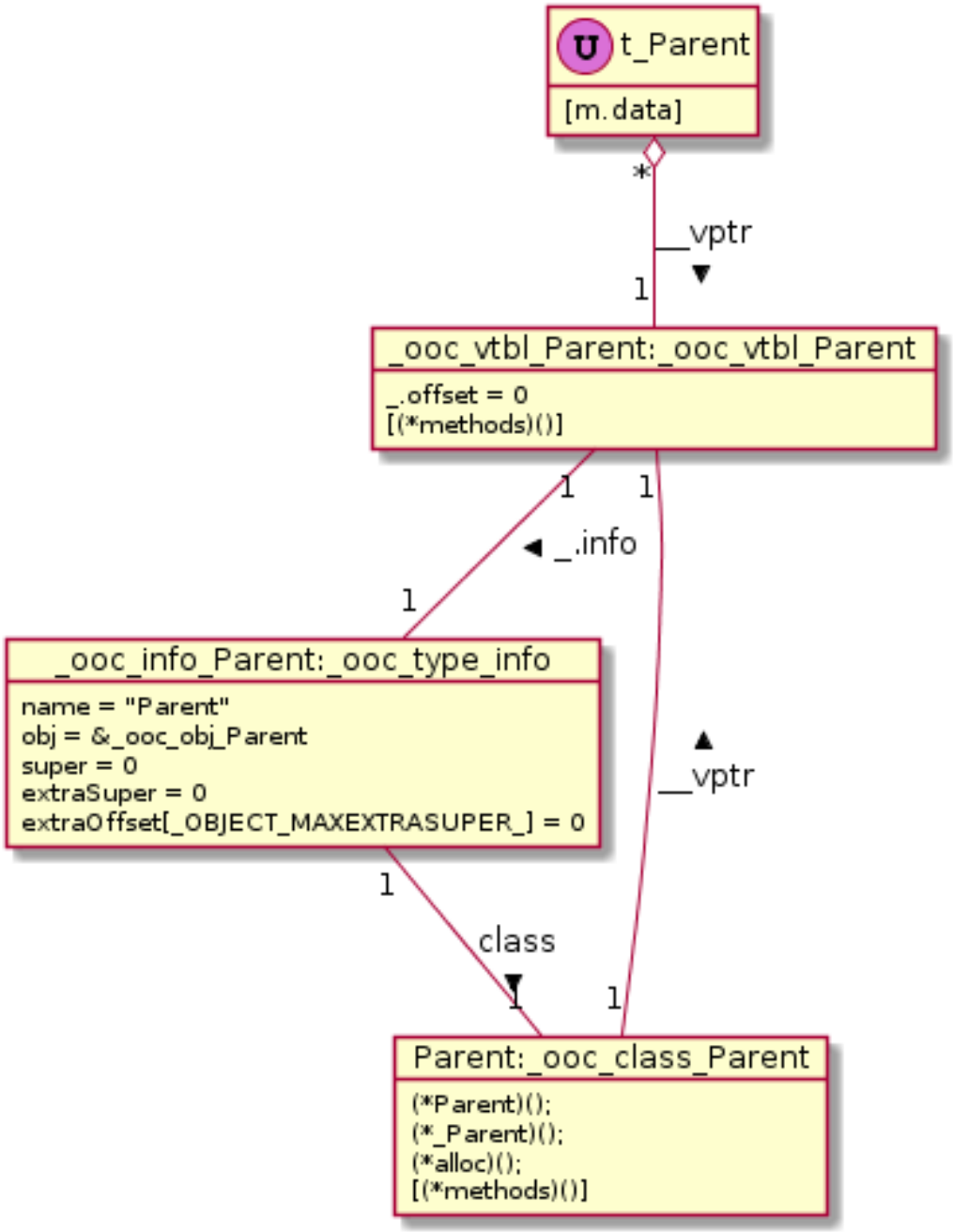
1. Encapsulamiento
2. Herencia (múltiple)
3. Polimorfismo
4. Genericidad
5. Excepciones

4.8.3 Modelo de objetos

Debido principalmente a la herencia múltiple, OOPC posee un modelo de objetos más complejo que los vistos anteriormente, por lo que se analizará con más de un diagrama.

4.8.3.1 Clase base

El siguiente diagrama representa el modelo de objetos que da soporte a todo el framework. Representa las estructuras e instancias que dan soporte a una clase `Parent` (el nombre `Parent` podría reemplazarse por cualquier otro nombre).



Analizaremos los cuatro elementos del diagrama:

* La unión, representado con el estereotipo `U`, `t_Parent` contiene todos los datos de instancia dentro de una estructura sin identificador de tipo y anidada `m` y una referencia a su tabla virtual mediante `__vptr`. Cada instancia de la clase `Parent` significará una instancia de la estructura

t_Parent.

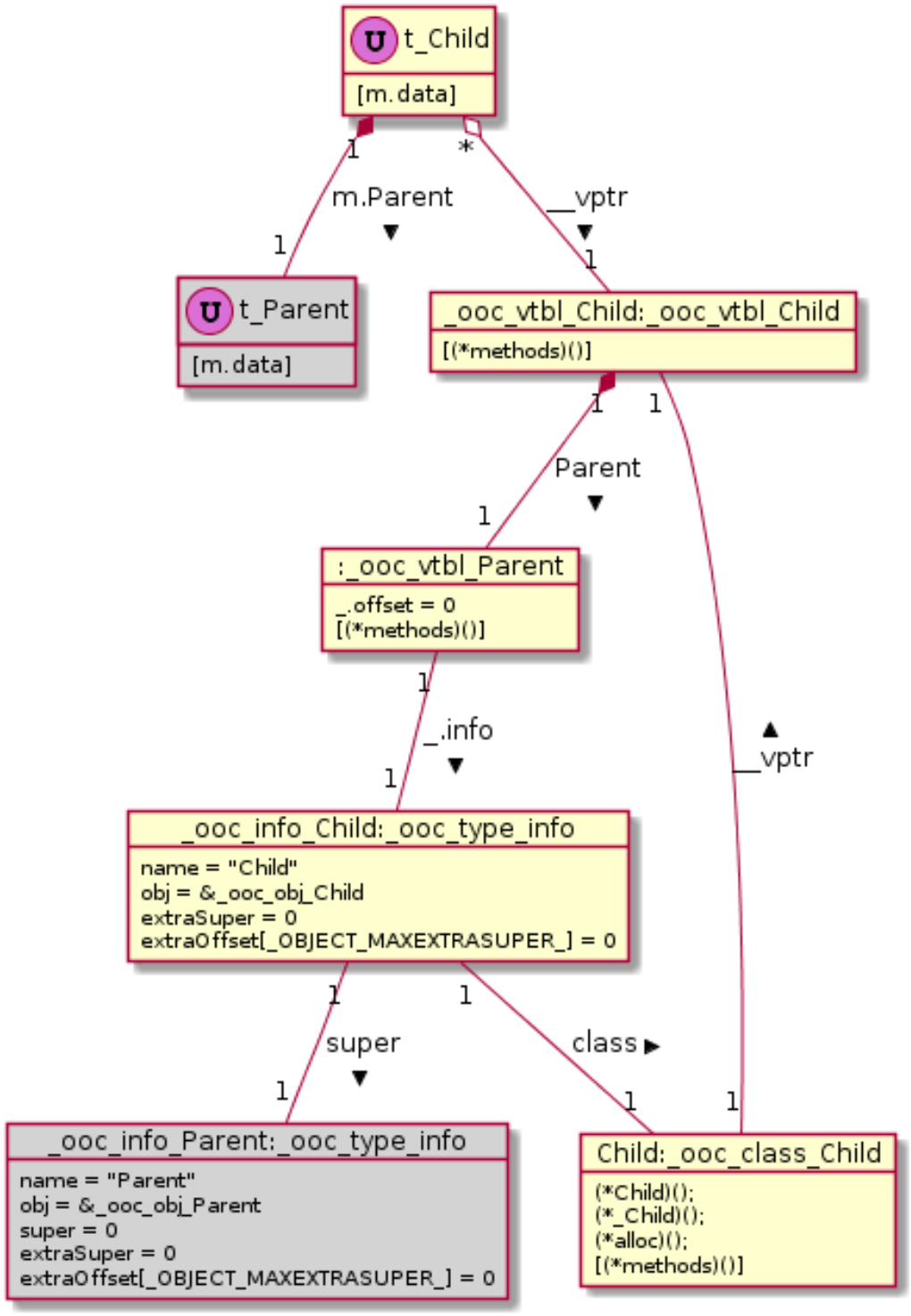
* La tabla virtual _ooc_vtbl_Parent es una instancia de la estructura _ooc_vtbl_Parent, y contiene los métodos polimórficos de instancia de la clase Parent. Contiene una instancia de la estructura _ooc_vtbl_object bajo el símbolo _, que contiene las variables offset e info. info es una referencia a la estructura _ooc_info_Parent que contiene la información de la clase Parent (RTTI). offset es utilizada, como veremos, para manejar la herencia múltiple.

* La estructura de información de tipo _ooc_info_Parent contiene la información de la clase Parent, su nombre en name, una instancia de t_Parent constante no inicializada pero con referencia a la tabla virtual en obj, y una referencia la estructura Parent.

* La estructura Parent es una instancia de _ooc_class_Parent, contiene los miembros de clase (atributos y métodos de clase) y métodos de instancia no polimórficos. Entre los métodos de clase generados están Parent() que inicializa la clase y retorna una instancia constante inicializada con valores por defecto y con su referencia a la tabla virtual (_ooc_obj_Parent), el destructor de instancia _Parent(), y el instanciador alloc() que retorna una copia en heap de _ooc_obj_Parent. Es común en esta estructura definir métodos de clase como new, copy e init. La referencia a la tabla virtual __vptr permite llamar a los métodos polimórficos de Parent en forma no polimórfica cuando el tipo del objeto es conocido, además permite que la clase misma (y no una instancia de la misma) sea tratada como un objeto (al contener métodos de clase polimórficos redefinibles por las clases heredadas).

4.8.3.2 Herencia simple

El siguiente diagrama muestra las estructuras que dan soporte a una clase Child que hereda de Parent.

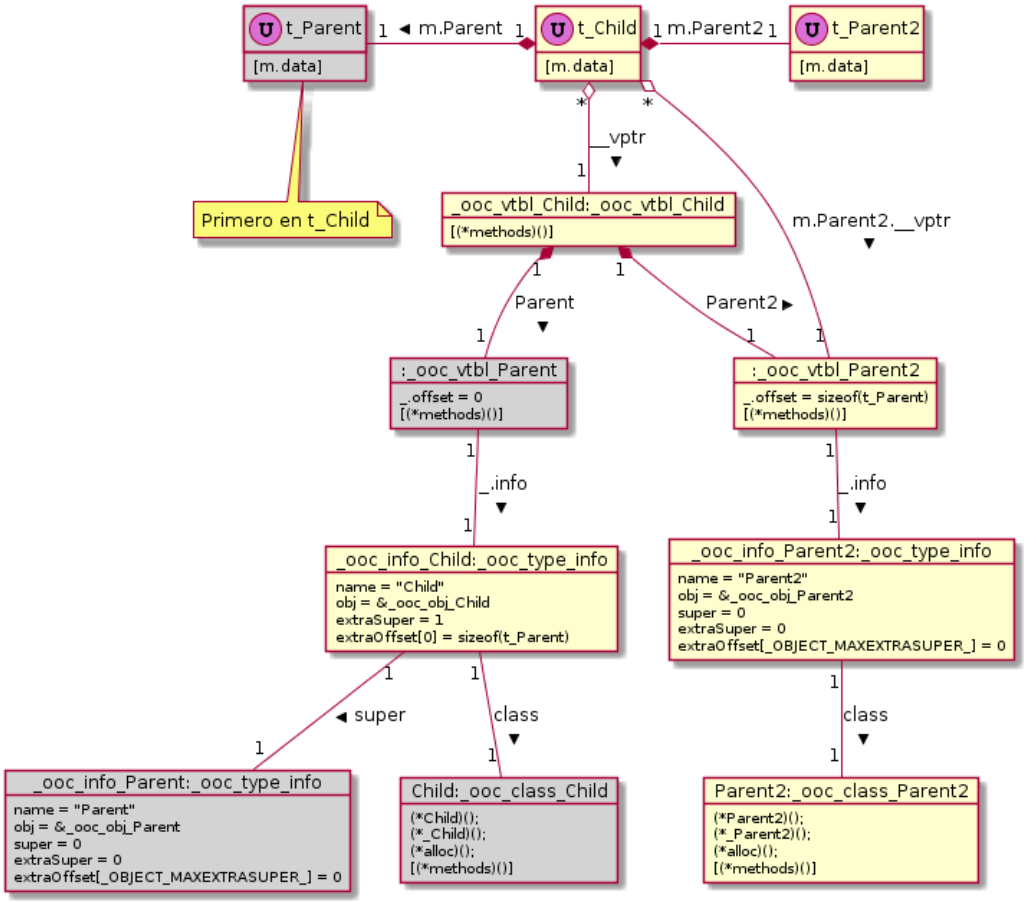


En gris se encuentran los elementos ya presentados para la clase `Parent`.
La unión t_Child incluye a la unión t_Parent como primer elemento.

Esto, como ya hemos visto en los frameworks anteriores, permite que las instancias de `Child` (o t_Child) puedan ser vistas como instancias de `Parent` (o t_Parent) para los métodos definidos para `Parent`.
La tabla virtual _ooc_vtbl_Child contiene la estructura de la tabla virtual de Parent (_ooc_vtbl_Parent) como primer elemento de su estructura. Esto también es necesario para que una instancia de `Child` pueda ser considerada una de `Parent` conteniendo los mismos métodos polimórficos (aunque pueden redefinirse para `Child`). Luego de la estructura _ooc_vtbl_Parent vienen los métodos polimórficos definidos especialmente para `Child`.
La estructura de RTTI es análoga a la de la clase base `Parent` pero el atributo `super` es inicializado con la estructura de RTTI de `Parent`. Esto permite realizar casteos dinámicos donde no se conoce la clase del objeto.

4.8.3.3 Herencia múltiple

El siguiente diagrama muestra cómo quedaría el modelo de objetos si `Child` heredara de una clase adicional `Parent2`. Las estructuras no afectadas respecto a la herencia simple aparecen en gris.



Podemos apreciar que la estructura `t_Child` está compuesta primero por la estructura `t_Parent` continuada por la estructura `t_Parent2`.
Cada una de sus referencias a una tabla virtual (la contenida en `t_Parent` y la contenida en `t_Parent2`) referencia a una tabla diferente. `t_Parent2` en `t_Child` referencia a una tabla virtual con la misma estructura a la de cualquier instancia de una clase `Parent2` y con la misma referencia a la estructura RTTI de un `Parent2`. Pero existe una gran diferencia con cualquier instancia de una clase `Parent2`, el atributo `offset` guarda donde se encuentra la estructura `t_Parent2` en `t_Child`, esta información permite que al pasar una referencia de `Parent2` en una instancia de `Child` (por ejemplo a una función que espera recibir un `Parent2` como parámetro) se pueda referenciar, a la instancia de `Child` que la contiene, en tiempo de ejecución (casteo dinámico). Por otro lado `t_Parent` en `t_Child` referencia a los mismas instancias que si tan solo hubiese una herencia simple entre

`t_Parent` y `t_Child`, la única diferencia es que en la estructura de RTTI se indica mediante el atributo `extraSuper` la cantidad de herencias por encima de la simple y en `extraOffset` la posición relativa de las mismas dentro de la estructura `t_Child`.

4.8.4 Codificación

OOPC facilita varias macros para la codificación y uso de las clases mediante la inclusión del archivo `ooc.h`

4.8.4.1 Codificación de una clase base

4.8.4.1.1 Interfaz (.h)

```
//Parent.h
#undef  OBJECT
#define OBJECT Parent

/* Object interface */
BASEOBJECT_INTERFACE

  char const* private(name);
  int private(attribute);

BASEOBJECT_METHODS

  void constMethod(print);

ENDOF_INTERFACE

/* Class interface */
CLASS_INTERFACE

  t_Parent*const classMethod_(new) char const name[] __;
```

```
  void method_(init) char const name[] __;
  void method_(copy) t_Parent const*const parent __;
  int class_attr;

ENDOF_INTERFACE
```

Cada clase comienza redefiniendo (mediante `#undef` y seguido por `#define`) la macro `OBJECT`, esto es necesario ya que el resto de las macros evalúan este valor.
Al tratarse de una clase base se utilizan las macros `BASEOBJECT_INTERFACE` y `BASEOBJECT_METHODS` en ese orden.
Debajo de la macro `BASEOBJECT_INTERFACE` se declaran las variables de instancia (`attribute` en el ejemplo) y luego de `BASEOBJECT_METHODS` los métodos polimórficos de instancia (`print`).
Debajo de `CLASS_INTERFACE` se declaran tanto los miembros de clase (variables (`class_attr`)y métodos (`new`)) como los métodos no polimórficos de instancia (`init` y `copy`). El método `print` es definido como un método de instancia constante usando la macro `constMethod()`, lo que significa que no modifica el estado del objeto al ejecutarse. La macro `method()` se utiliza para métodos no constantes y para los métodos de clase (que no referencian a ningún objeto) `classMethod()`. Cada uno de ellos tiene su equivalente para métodos con argumentos agregando un `_` al final (`method_()` `constMethod_()` `classMethod_()`), si nos encontramos en C99 dentro de los paréntesis podemos agregar los argumentos (por ejemplo `method_(init,char const name[])`), pero bajo C89 los argumentos van a continuación terminados con `__` como en el ejemplo (`method_(init) char const name[] __`).
Como puede apreciarse, el atributo `attribute` tiene visibilidad privada mediante la macro `private()`, esta macro puede utilizarse tanto para variables como para métodos. Esto se logra cambiando el nombre del argumento por algo indicativo de ser privado (en el ejemplo attribute___ooc_private_40201112L), pero no cambiándolo en el archivo de implementación o código fuente donde se define la macro `IMPLEMENTATION` como se verá a continuación.

4.8.4.1.2 Implementación (.c)

```
//Parent.c
// include de otros módulos
#define IMPLEMENTATION

#include <Parent.h>

void
constMethodDecl(print)
{
  // print implementation
}

BASEOBJECT_IMPLEMENTATION

  methodName(print)

ENDOF_IMPLEMENTATION

/*
  --------------------
  Class implementation
  --------------------
*/

initClassDecl() /* requerido */
{
    /* código de inicialización de la clase */
    objDefault(attribute) = 1;
}

dtorDecl() /* requerido */
{
  /* código del destructor de instancias */
```

```
}

t_Parent
classMethodDecl_(*const new) char const name[] __
{
  /* código de método new */
  return this;
}

void
methodDecl_(init) char const name[] __
{
  /* código de método init */
}

void
methodDecl_(copy) t_Parent const*const per __
{
  /* código de método copy */
}

CLASS_IMPLEMENTATION

  methodName(new),
  methodName(init),
  methodName(copy),
  0 /*class_attr=0*/
ENDOF_IMPLEMENTATION
```

El archivo implementación comienza con la inclusión de otros módulos y clases, esto es así para que no sean afectadas por la definición de la macro `IMPLEMENTATION` que es definida a continuación seguida por la inclusión del archivo de interfaz (.h). La declaración de los métodos para su implementación se realiza mediante una macro paralela a la que se usa para la declaración en la interfaz.

implementacion	interfaz
classMethodDecl()	classMethod()
classMethodDecl_()	classMethod_()
methodDecl()	method()
methodDecl_()	method_()
constMethodDecl()	constMethod()
constMethodDecl_()	constMethod_()

Entre `BASEOBJECT_IMPLEMENTATION` y `ENDOF_IMPLEMENTATION` se encuentran los métodos de instancia (en el mismo orden que se declararon en el interfaz), por lo que antes de estas macros deben declararse.
Entre las macros `CLASS_IMPLEMENTATION` y `ENDOF_IMPLEMENTATION` se encuentran los miembros de clase (métodos e inicialización de atributos), por lo que antes de estas macros deben declararse sus métodos. `initClassDecl()` y `dtorDecl()` son macros obligatorias en la implementación de una clase. `initClassDecl()` se ejecuta una sola vez al instanciarse el primer objeto de la clase y se utiliza en una clase base tan solo para inicializar atributos de clase y asignar valores por defecto de los atributos de instancia (mediante la macro `objDefault` que cambia los atributos de `_ooc_obj_Parent`), mientras que `dtorDecl()` se llama cada vez que un objeto de la clase es destruido (mediante la macro `delete()` si se utiliza memoria dinámica o con la llamada explícita al destructor, en el ejemplo `Parent._Parent()`).

4.8.4.2 Codificación de una clase derivada

Siguiendo el ejemplo de modelo de objetos presentamos el código de una clase `Child` que hereda de `Parent` y de `Parent2`.

4.8.4.2.1 interfaz (.h)

```
//Child.h
#include <Parent.h>
```

```
#include <Parent2.h>

#undef  OBJECT
#define OBJECT Child

/* Object interface */
OBJECT_INTERFACE

  INHERIT_MEMBERS_OF (Parent);
  INHERIT_MEMBERS_OF (Parent2);
  int private(child_attribute);

OBJECT_METHODS

  INHERIT_METHODS_OF (Parent);
  INHERIT_METHODS_OF (Parent2);
  void method(child_method);

ENDOF_INTERFACE

/* Class interface */
CLASS_INTERFACE

  t_Child*const classMethod_(new)
     char const name[]__;
  void method_(init)
     char const name[]__;
  void method_(copy) t_Child const*const child __;

ENDOF_INTERFACE
```

Podemos apreciar que debajo se las macros `OBJECT_INTERFACE` y `OBJECT_METHODS` se declara la herencia de `Parent` y de `Parent2` (mediante las macros `INHERIT_MEMBERS_OF` y `INHERIT_METHODS_OF` en el orden de la herencia). Luego de los mismos continúa la declaración de atributos y de

métodos propios de `Child`. La declaración de los miembros de clase es la misma que la de una clase base.

4.8.4.2.2 Implementación (.c)

```
//Parent.c
#define IMPLEMENTATION

#include "Child.h"

void
methodDecl(child_method)
{
  /*código para child_method()*/
}

void
constMethodOvldDecl(print, Parent)
{
  /*código para redefinición de print()*/
}

OBJECT_IMPLEMENTATION

  SUPERCLASS (Parent),
  SUPERCLASS (Parent2),
  methodName(child_method)

ENDOF_IMPLEMENTATION

initClassDecl()
{
  /* inicializar clases padre */
  initSuper(Parent);
  initSuper(Parent2);
```

```
  /* redefinición de métodos */
  overload(Parent.print) =
      methodOvldName(print, Parent);


}

dtorDecl()
{
    /*Liberación de recursos de Child*/
    Parent._Parent(super(this,Parent));
    Parent2._Parent2(super(this,Parent2));
}

/*...*/

void
methodDecl_(init)
     char const name[]__
{
  Parent.init(super(this,Parent), name);
  Parent2.init(super(this,Parent2));
  /*más código de inicialización*/
}

CLASS_IMPLEMENTATION

  methodName(new),
  methodName(init),
  methodName(copy)

ENDOF_IMPLEMENTATION
```

Para redefinir métodos se utilizan las macros especiales para su declaración.

La siguiente tabla muestra las macros para las declaraciones en las clases base y su equivalente para su redefinición en las clases heredadas:

clase base	clase heredada
methodDecl()	methodOvldDecl()
methodDecl_()	methodOvldDecl_()
constMethodDecl()	constMethodOvldDecl()
constMethodDecl_()	constMethodOvldDecl_()

En el código del ejemplo podemos apreciar la redefinición del método `print()` con la macro `constMethodOvldDecl()`, que luego se debe asignar en la inicialización de la clase a su referencia en la tabla virtual mediante las macros `overload` y `methodOvldName` (en el ejemplo la línea: `overload(Parent.print) = methodOvldName(print, Parent);`). Debajo de la macro `OBJECT_IMPLEMENTATION` indicamos la herencia de las clases padre mediante la macro `SUPERCLASS`. Luego en la inicialización de la clase, bajo la macro `initClassDecl()`, inicializamos las instancias de la clases padre dentro de la clase base (dentro de las estructuras de la tabla virtual y de RTTI), además se actualizan los valores de `extraSuper` y `extraOffset` (ver modelo de objetos).
Al final del destructor de `Child()` (`dtorDecl()`) se llama a los destructores de las clases padre.
En el ejemplo donde creamos los métodos `init()` para la inicialización de instancias de cada clase, el método `init()` de `Child` llama a los de sus clases padre al principio.

4.8.4.3 Clase abstracta

Una clase abstracta no puede instanciarse y puede no proveer una definición de sus métodos. Esto se logra parcialmente en OOPC proveyendo la macro `ABSTRACTCLASS_INTERFACE` para ser utilizada en vez de `CLASS_INTERFACE` y cuya única diferencia es que no define al método `alloc()`. Luego las asignaciones a las referencias a métodos mediante las macros `methodName()` y `methodOvldName()` de los métodos que no proveen una implementación

deben ser reemplazadas por 0.

4.8.4.4 Clase genérica

Para codificar clases genéricas OOPC utiliza un mecanismo simple valiéndose del preprocesador de C. Todos los tipos genéricos se declaran como `gType1`, `gType2`, etc. postergando su definición a la implementación del usuario (también podrían usarse otros nombres como `T`). Luego para cada clase genérica que el usuario instancie en una clase concreta deberá especificar un prefijo para dicha clase mediante la definición de la macro `gTypePrefix`.

4.8.4.4.1 Instanciación El siguiente es un ejemplo de instanciación de una clase genérica tanto para un tipo entero como uno flotante en los mismos archivos .c y .h

```
//array.h
/* integer array */

#define gTypePrefix i
#define gType1      int

#include <g_array.h>

#undef gType1
#undef gTypePrefix

/* double array */

#define gTypePrefix d
#define gType1      double

#include <g_array.h>
```

```
#undef gType1
#undef gTypePrefix

//array.c
/* integer array */

#define gTypePrefix i
#define gType1      int

#include <g_array.c>

#undef gTypePrefix
#undef gType1

/* double array */

#define gTypePrefix d
#define gType1      double

#include <g_array.c>

#undef gTypePrefix
#undef gType1
```

4.8.4.4.2 Declaración El siguiente es un ejemplo de la interfaz de una clase genérica.

```
//g_array.h
#if defined(GENERIC) || !defined(G_ARRAY_H)
#define G_ARRAY_H
/*...*/
#undef  OBJECT
#define OBJECT GENERIC(array)
```

```
/* Object interface */
OBJECT_INTERFACE

  INHERIT_MEMBERS_OF(GENERIC(memBlock));

  gType1* private(data);

OBJECT_METHODS

  INHERIT_METHODS_OF(GENERIC(memBlock));

  gType1* method(getData);
  /*...*/
ENDOF_INTERFACE


CLASS_INTERFACE
    /*...*/
ENDOF_INTERFACE

#endif
```

La macro `OBJECT` se define con la macro `GENERIC` que es la encargada de agregar el prefijo a la instanciación. La clase genérica array desciende de la clase genérica memblock (para facilitar la referencia al destructor de la clase padre se utiliza la macro `GENERIC_DTOR(memblock)`). El tipo genérico `gType1` es luego definido por el usuario.

4.8.5 Dificultades en la codificación

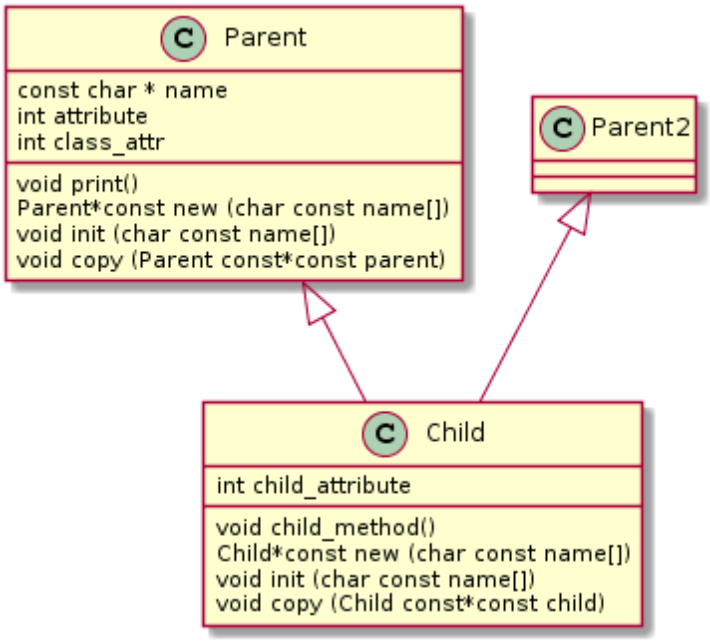
#	Tipo de dificultad	Nombre	Descripción
1	Propensión a errores	Inicialización de métodos polimórficos	Contrario a otros LPOO a la definición de un método debe asignársele su implementación en la función de inicialización de la clase (bajo la macro `initClassDecl()`), si esto se olvida generaremos un error crítico al enviar un mensaje con el método no inicializado.
2	Propensión a errores y obtención de información	Prototipado de métodos polimórficos redefinidos	Para crear el prototipo de un método ya definido por una clase padre se debe conocer exactamente cuál de los padres es el que definió el método por primera vez, además de que en caso de equivocarnos recibiremos una advertencia por parte del compilador, esto agrega una demora al programador que debe obtener esta información.
3	Propensión a errores y obtención de información	Asignación en la tabla virtual de métodos polimórficos redefinidos	Para asignar la reimplementación de un método en la tabla virtual a través de la macro `overload()` se debe conocer toda la jerarquía de clases hasta la clase que definió por primera vez el método, además de la posible equivocación de cálculo esto demora tiempo al programador para obtener dicha información. Además si nos encontramos en el contexto del problema del diamante deberemos repetir esto por cada uno de los caminos de ascendencia.

#	Tipo de dificultad	Nombre	Descripción
4	Repetición de información	Cada método implica de 3 referencias al mismo	Incluirlo en la estructura de la clase o la tabla virtual (dependiendo si es polimórfico o no), en su implementación, asignar la implementación al puntero a función de la estructura que lo representa.
5	Repetición de información	Repetición en la información de la herencia	En la interfaz se duplica la información de la herencia de clases (bajo las macros`INHERIT_MEMBERS_OF` y `INHERIT_METHODS_OF`), lo mismo pasa en el archivo de implementación en cada redefinición de un método, en el método del destructor y bajo la macro `OBJECT_IMPLEMENTATION`.
6	Propensión a errores	Destructor de instancias	No se debe olvidar para el destructor de una clase heredada llamar al destructor de la clase padre **al final** del destructor.
7	Propensión a errores	Orden de los métodos	Para la definición de métodos (y no su redefinición que se realiza bajo la macro `initClass()`) el orden de los métodos debajo de `BASEOBJECT_METHODS` debe ser el mismo que debajo de `OBJECT_IMPLEMENTATION`. En caso de haber invertido el orden y el prototipo de los métodos invertidos es el mismo, no nos lo advertirá el compilador.

#	Tipo de dificultad	Nombre	Descripción
8	Repetición de información	Duplicación en la redefinición de métodos	Se duplica la información de la redefinición de un método (con la macro `constMethodOvldDecl()` y la asignación mediante las macros `overload` y `methodOvldName`).
9	Dificultad de aprendizaje	Gran cantidad de nuevas macros	OOPC no es un framework minimalista como los anteriores, hay una gran cantidad de macros que utilizar y aprender.
10	Repetición de código	Repetición en cada clase	Comparado con otros LPOO, hay mucho código repetido entre una clase y otra tan solo para definir la clase y su herencia.
11	Propensión a errores	Macros parecidas para distintos contextos en la implementación de métodos	Por ejemplo, para un método declarado en la interfaz con `method()` luego se lo implementa con `methodDecl()` y se lo asigna en la tabla virtual o estructura de clase con `methodName()`, luego para su redefinición en una clase derivada se utiliza `methodOvldDecl()` y la asignación en la tabla virtual se realiza con `methodOvldName()`, muy fácilmente podemos confundirnos entre ellos.

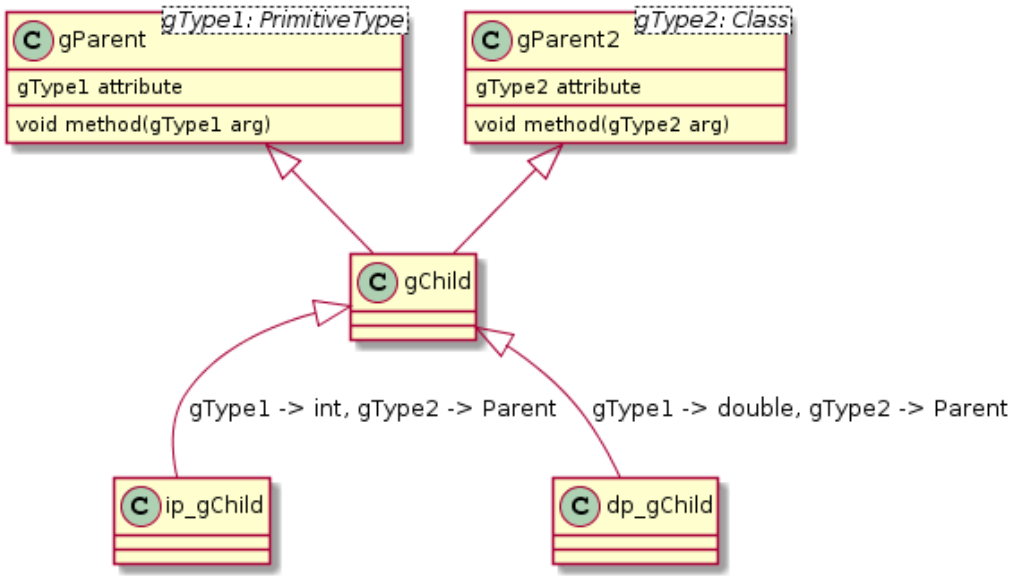
4.8.6 Propuesta de expresión en UML

En la siguiente figura representamos el diagrama de clases UML para el ejemplo de codificación de herencia múltiple.



De acuerdo al manual de referencia de UML (OMG 2017), los atributos y métodos privados se marcan con el atributo `visibility` en `private`, y los públicos con `public`, si son constantes entonces el atributo `isReadOnly` debe ser `true` y si son de clase `isStatic` (por ejemplo `class_attr`). Los métodos constantes contienen el atributo `isReadOnly` (por ejemplo `print()`) en `true`, los abstractos `isAbstract` (no se genera implementación para el método), los no polimórficos `isLeaf` (por ejemplo `init()` y `copy()`) y los de clase `isStatic` (por ejemplo `new()`). Además, las clases pueden ser abstractas (no proveen el método `alloc()`) con el atributo `isAbstract` en `true` este podría ser el caso por ejemplo de `Parent2` (manteniendo el ejemplo de codificación dado). Todo esto se puede especificar para OOPC.

El siguiente diagrama representa la propuesta de representación en UML de las clases genéricas en OOPC.



La clases genéricas pueden heredarse por otras. También pueden ser instanciadas en clases concretas asignando un tipo concreto a sus tipos genéricos. En el ejemplo, la relación entre `gChild` y `ip_gChild` o `dp_gChild` no es de herencia sino de vinculación de elementos genéricos (template binding), donde se asigna el tipo concreto. Las clases vinculantes (`ip_gChild` y `dp_gChild`) no pueden declarar miembros nuevos, pero sí lo podría hacer una clase que herede de cualquiera de ellas (esto permite no tener que obligar esa herencia para permitir la declaración de nuevos miembros como lo permite UML). Las clases vinculantes deben terminar con el nombre de la clase genérica siendo el prefijo restante el asignado a la macro `gTypePrefix`. Ninguna de las dificultades enumeradas en la sección anterior aplican para estos diagramas.

4.9 Object-Oriented C de Tibor Miseta (ooc)

4.9.1 Introducción

El propósito de OOC es facilitar la POO en microcontroladores pequeños y para que el programador C aprenda conceptos de orientación a objetos.

Está escrito bajo ISO C89. Junto con la biblioteca del sistema de objetos y macros para facilitar la codificación, provee una herramienta para facilitar la creación de clases, interfaces y mixins desde templates u otras clases ya implementadas. El autor escribe acerca de la codificación de estos artefactos:

"Crear clases en ooc escribiéndolas desde cero puede requerir mucho trabajo, es propenso a errores, pero principalmente aburrido"[Tibor2017]. Lamentablemente esta herramienta no facilita la inicialización de la tabla virtual ni la implementación de miembros de clase o instancia como sí lo puede hacer un generador de código desde UML.

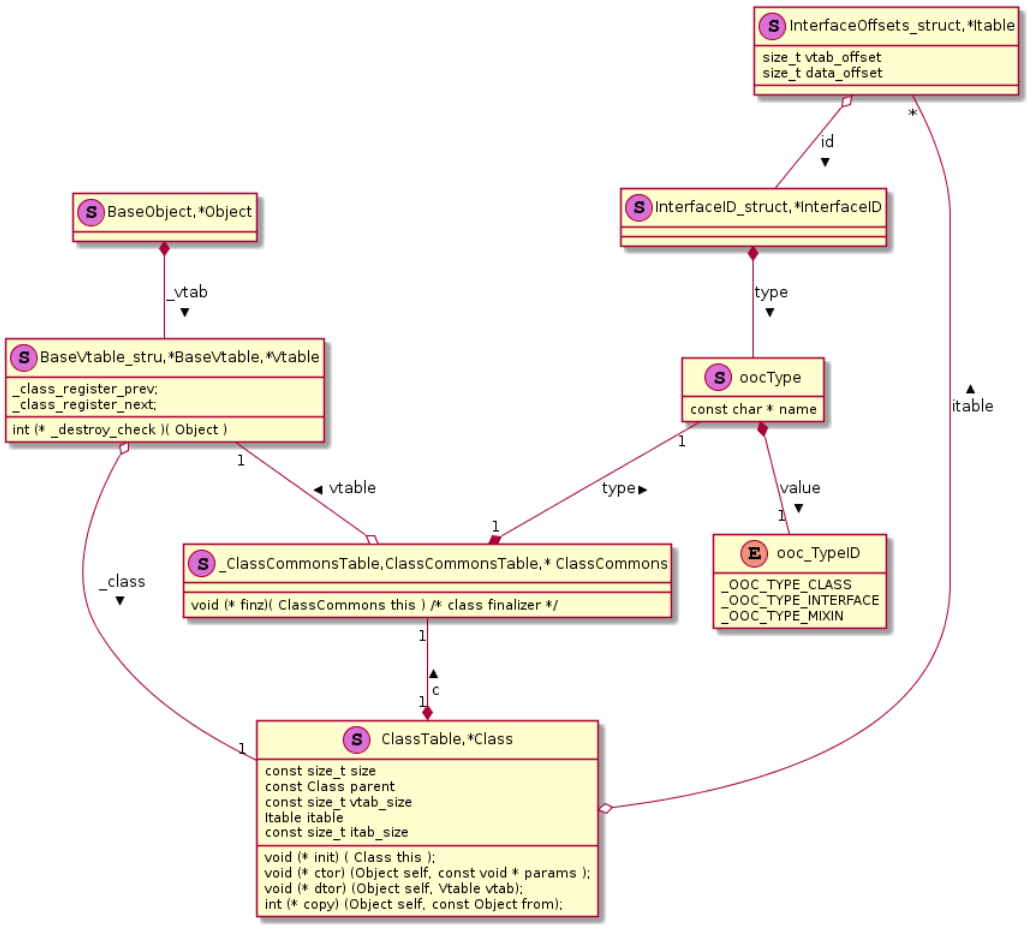
4.9.2 Conceptos soportados

1. Encapsulamiento
2. Herencia
3. Polimorfismo
4. Interfaces (sin herencia de interfaces)
5. Mixins (sin herencia de mixins)
6. Excepciones

4.9.3 Modelo de objetos

4.9.3.1 Clases

El siguiente diagrama nos muestra las estructuras que dan soporte a las clases en ooc.

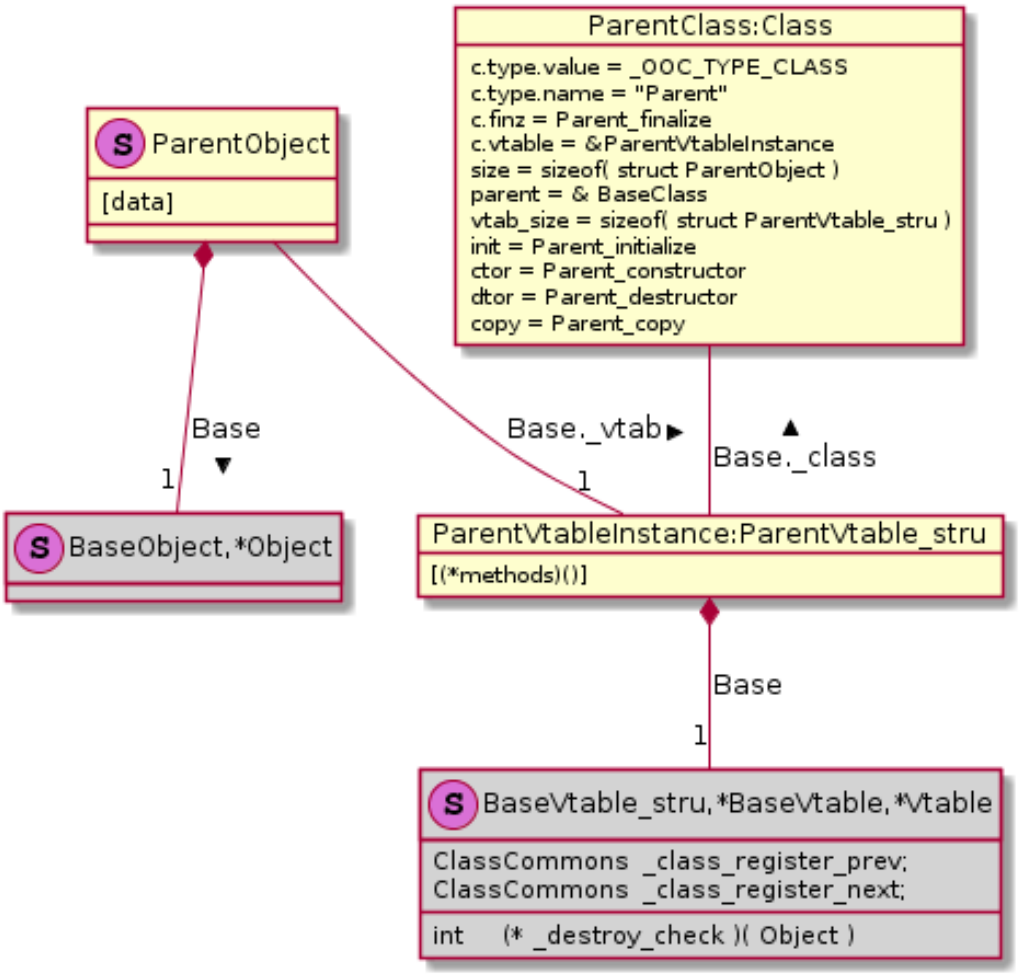


La estructura base de la cual todas las estructuras de instancia de clases heredan es `BaseObject`. Como en los frameworks anteriores, la herencia se consigue incluyendo a la estructura de la clase padre como primer miembro de la estructura de la clase derivada.

`BaseObject` contiene una referencia a una tabla virtual (con estructura `BaseVtable_stru`), de las cuales también heredan las tablas virtuales de otras clases para declarar sus métodos polimórficos. La tabla virtual referencia a la estructura que representa a la clase y que sirve de estructura de RTTI, contiene los constructores y destructores de clase (para instanciar y liberar la clase) y de instancia. El nombre de la clase y su tipo (para una clase `_OOC_TYPE_CLASS`). También incluye un listado de interfaces y mixins (qué es un mixin se explicará a continuación) a través de la referencia al arreglo itable, el mismo nos dice donde se encuentran los métodos

definidos para la interfaz dentro de la tabla virtual (a través del atributo `vtab_offset`) y en caso de tratarse de un mixin, dónde se encuentran sus datos dentro de la estructura de la instancia de clase (a través del atributo `data_offset`). Qué mixin o interfaz se está implementando se asigna en la referencia `id`.

4.9.3.1.1 Clases Base Para entender como funciona el modelo de objetos veremos las estructuras e instancias que se crean para una clase base de ejemplo (`Parent`). Las estructuras que ya introducimos aparecen en gris.



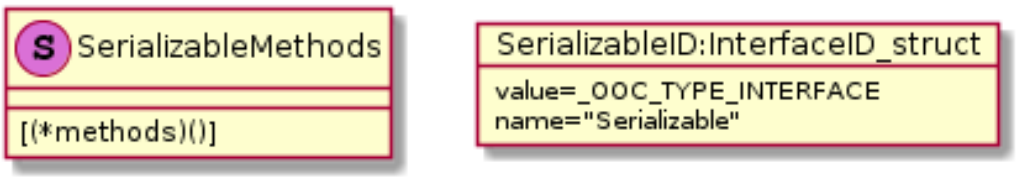
Como vemos en el diagrama hay dos herencias de estructuras. Una de `ParentObject` con `Object` de la cual hereda su referencia a una tabla virtual y se extiende declarando sus propias variables de instancia. Y otra de

`ParentVtable_str` con `BaseVtable_str` que le permite usarse como una tabla virtual y extenderse definiendo nuevos métodos de instancia y métodos y atributos de clase. Esta tabla virtual referencia a su propia instancia de la estructura Class donde se definen miembros comunes a todas las clases. El atributo `parent` de la clase, que referencia a la clase padre de la clase `Parent`, referencia a `BaseClass` lo que indica que esta es una clase base. Una clase que herede de `Parent` directamente tendría una referencia a `ParentClass` en este atributo.

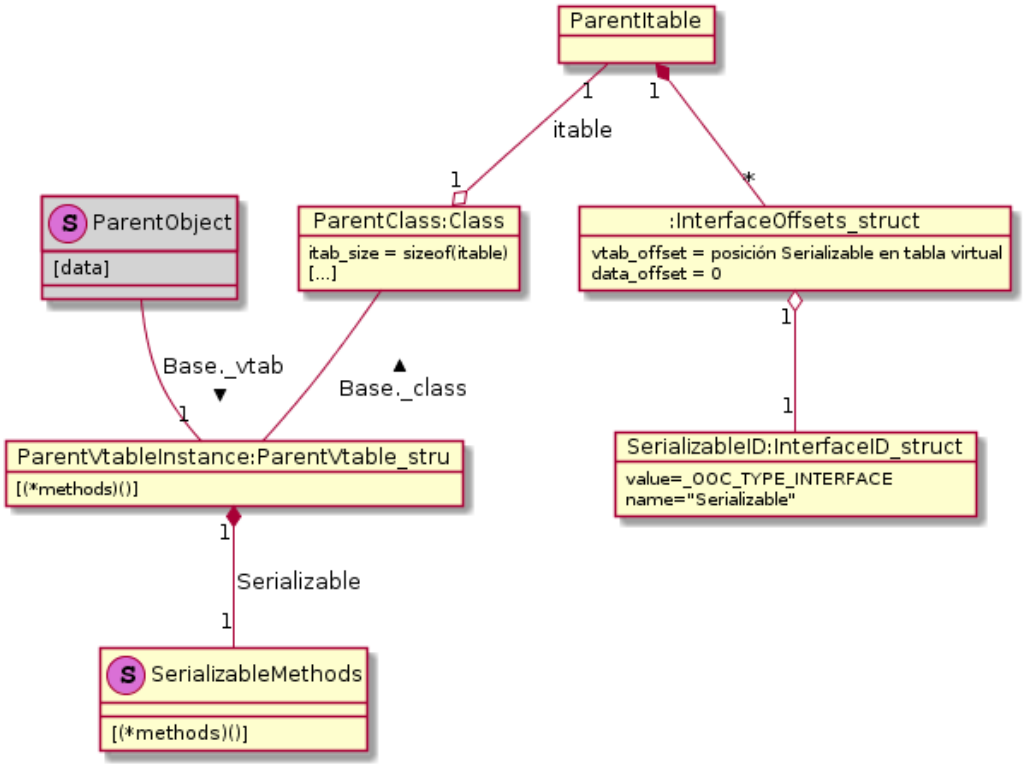
Esta clase no realiza interfaces o hereda mixins, los que veremos a continuación.

4.9.3.2 Interfaces

Las estructuras en instancias que dan soporte a una interfaz son muy sencillas. El siguiente diagrama nos muestra un ejemplo con una interfaz llamada `Serializable`

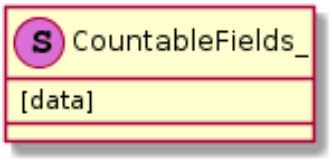


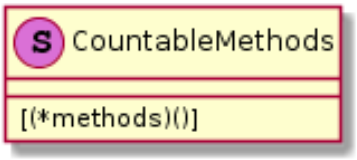
La instancia de `InterfaceID_struct` sirve para identificar a las instancias de `SerializableMethods` como tales. Si deseamos realizar esta interfaz en la clase `Parent` deberíamos modificar su tabla virtual y estructura de RTTI. El siguiente diagrama nos muestra estas modificaciones.

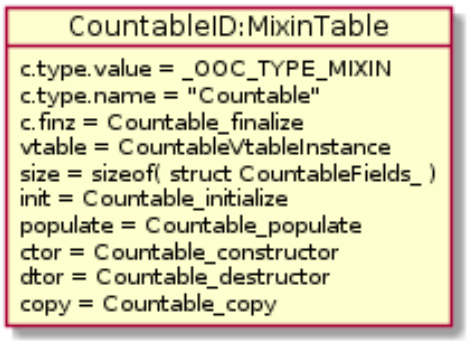


4.9.3.3 Mixins

Los mixins son muy parecidos a las clases pero en vez de instanciarse, son heredados por una clase. A la vez agregan una interfaz a la clase contenedora del mixin por lo que son como una Interfaz que define parcial o totalmente su implementación. El siguiente diagrama nos muestra las estructuras e instancias que dan soporte a un mixin llamado `Countable` (podría llamarse de cualquier otra forma).



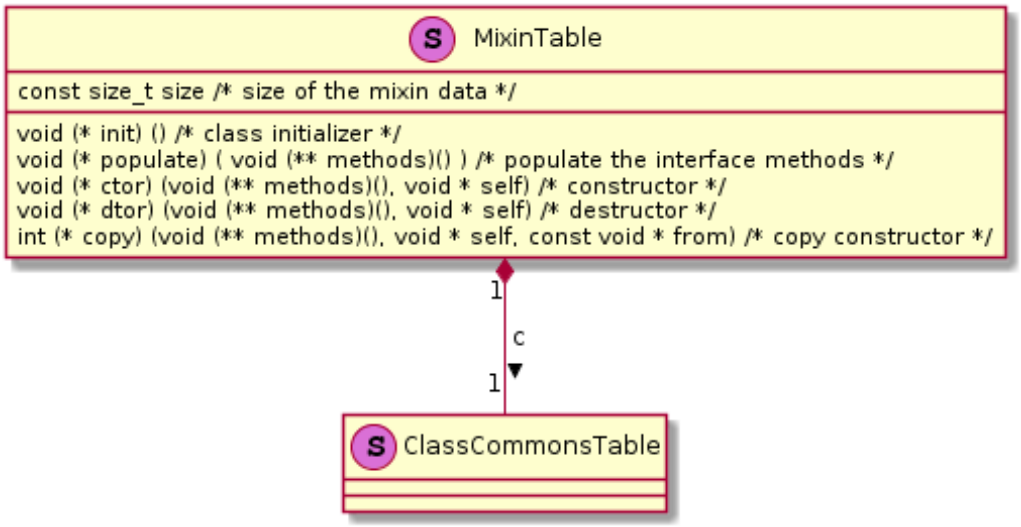




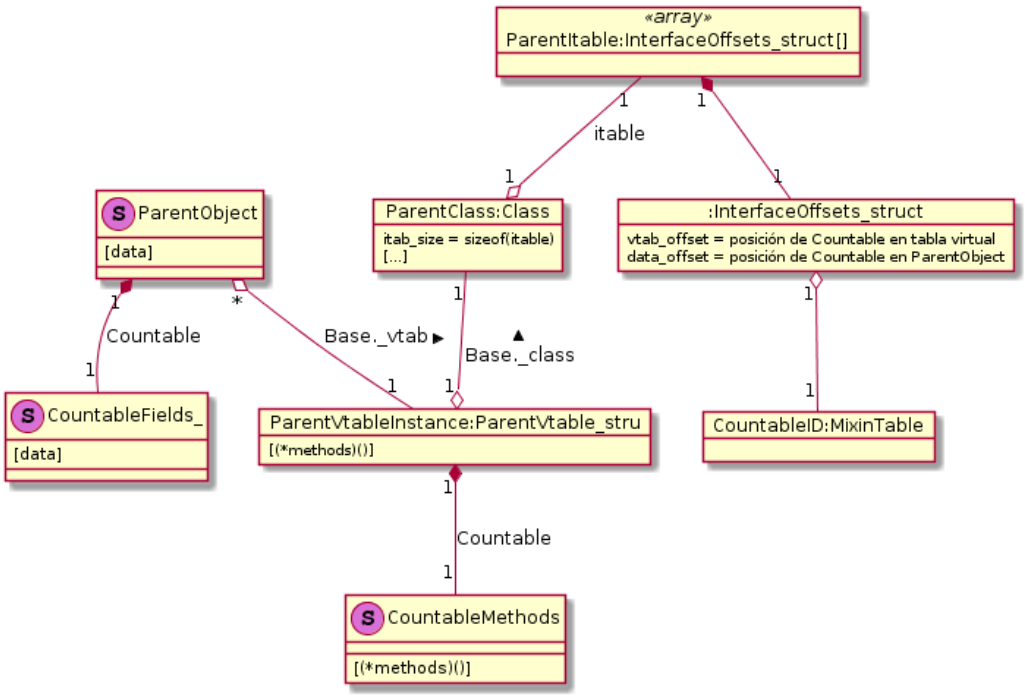
Al igual que una interfaz, se define una estructura terminada en "Methods" (en nuestro caso `CountableMethods`) que a su vez define las referencias a los métodos definida para el mixin, y a diferencia de las interfaces se define una estructura de variables (`CountableFields_`) definida para el mixin. La instancia de `MixinTable` (`CountableID`) es análoga a la estructura `ClassTable` de una clase. Incluye la información RTTI del Mixin. Esto permite identificar las instancias de las estructuras `CountableMethods` y `CountableFields_` al asociarlas con `CountableID`. Al igual que una clase, un mixin contiene, métodos de inicialización y finalización del mixin, y métodos de inicialización, copia y finalización de instancia. Las instancias de mixins son componentes de las clases que las heredan.

Un método `populate()` que es el encargado de inicializar la tabla virtual de la clase que instancia al mixin con los métodos implementados por el mixin. Los métodos implementados por el mixin pueden ser redefinidos por la clase que la hereda.

La estructura MixinTable hereda de ClassCommonsTable como se representa en el siguiente diagrama.



El siguiente diagrama nos muestra cómo quedarían las estructuras de `Parent` al heredar el mixin `Countable`



La estructura `ParentObject` contiene los atributos de instancia. También contiene los atributos definidos por el mixin bajo el nombre `Countable`. La tabla virtual de `Parent` contiene los métodos definidos por el mixin

también bajo el nombre `Countable`, al igual que con una interfaz. El arreglo `ParentItable` contiene una instancia de la estructura `InterfaceOffsets_struct`, la misma relaciona a la tabla que describe al mixin (`CountableID`) con la posición del mixin dentro de la tabla virtual (`vtab_offset`) y la posición del mixin dentro de las variables de instancia (`data_offset`). De esta manera desde la estructura de RTTI de la clase `Parent` se puede obtener.

4.9.4 Codificación

4.9.4.1 Visibilidad

La visibilidad de las clases y mixins en ooc son de dos tipos: de declaración (encapsulación fuerte) o de implementación y dependen del archivo de interfaz que incluyamos. Los dos archivos de interfaz, por convención, tienen el nombre de la clase (por ejemplo `clase.h`). La interfaz para obtener visibilidad de implementación se encuentra, por convención, en la carpeta `implement`. Las variables de instancia son accesibles bajo la visibilidad de implementación. Las clases heredadas acceden, en su implementación, a sus clases padres bajo la visibilidad de implementación.

4.9.4.2 Clases

4.9.4.2.1 Interfaz de declaración Las clases se declaran a través de la macro `DeclareClass`. El primer parámetro de la macro es la clase a declarar y la segunda la clase de quién hereda. Si se trata de una clase base se debe utilizar como segundo argumento de `DeclareClass` la palabra `Base`:

```
//Parent.h
DeclareClass( Parent, Base );
```

La tabla virtual se declara con con las macros `Virtuals` y `EndVirtuals`. Contiene la declaración de métodos virtuales, variables de clase y los métodos

de las interfaces (tanto interfaces comunes como mixins) a través de la macro `Interface`:

```
//Parent.h
Virtuals( Parent, Base )
    void    (* method)  ( Parent );
    int     classVariable;
    Interface( MyMixin );
    Interface( MyInterface );
EndOfVirtuals;
```

4.9.4.2.2 Interfaz de implementación Las variables de instancia y así como las variables heredadas de los mixins (a través de la macro `MixinData`) se declaran con las macros `ClassMembers` y `EndOfClassMembers`:

```
//implement/Parent.h
ClassMembers( Parent, Base )
    int     instanceVariable;
    MixinData(MyMixin);
EndOfClassMembers;
```

4.9.4.2.3 Implementación (.c) Las posiciones de las interfaces en la tabla virtual son registradas en un vector especializado mediante las macros `InterfaceRegister`, `AddMixin` y `AddInterface` :

```
//Parent.c
InterfaceRegister( Parent )
{
    AddMixin( Parent, MyMixin ),
    AddInterface( Parent, MyInterface )
};
```

La tabla virtual es instanciada mediante la macro `AllocateClassWithInterface`, en caso de que la clase no realice interfaces o mixins se utiliza la macro `AllocateClass`:

```
//Parent.c
AllocateClassWithInterface( Parent, Base );
```

Existen una variedad de funciones que deben implementarse para cada clase: inicialización y finalización de la clase, constructor y destructor de instancia y constructor de copias. En la inicialización de la clase se debe inicializar la tabla virtual con la implementación de los métodos declarados en la misma (sólo para los métodos en los que la clase defina o especialice al método, los métodos heredados son automáticamente inicializados en la tabla virtual por los ancestros de la clase):

```
static
void
Parent_initialize( Class this )
{
    ParentVtable vtab = & ParentVtableInstance;
    ((ParentVtable)vtab)->method = Parent_method;
    ((ParentVtable)vtab)->MyInterface.myInterfaceMethod =
                (void(*)(Object self))Parent_myInterfaceMethod;
```

4.9.4.3 Interfaces

Las interfaces no contienen variables de instancia por lo que no contienen un archivo header adicional.

4.9.4.3.1 Interfaz (.h) La declaración de la interfaz se realiza con la macros `DeclareInterface` y `EndOfInterface` declarando los métodos de la misma:

```
//MyInterface.h
DeclareInterface( MyInterface  )
    void (*myInterfaceMethod)(Object self);
EndOfInterface;
```

4.9.4.3.2 Implementación(.c) Por último se debe instanciar la estructura de RTTI de la interfaz en cualquier archivo, por ejemplo uno particular para la interfaz:

```
//MyInterface.c
AllocateInterface( MyInterface );
```

4.9.4.4 Mixins

Se componen de los mismos tres archivos que las clases (o sea uno de implementación y dos de interfaz).

4.9.4.4.1 Interfaz de declaración Para declarar la interfaz del mixin se utiliza la macro `DeclareInterface` y `EndOfInterface` declarando los métodos de la misma:

```
//MyMixin.h
DeclareInterface( MyMixin  )
    unsigned int (*myMixinMethod)(Object self);
EndOfInterface;
```

4.9.4.4.2 Interfaz de implementación. Los datos miembros del mixin se declaran entre las macros `MixinMembers` y `EndOfMixinMembers`:

```
//implement/MyMixin.h
MixinMembers( MyMixin )
    unsigned int instanceNum;
EndOfMixinMembers;
```

4.9.4.4.3 Implementación (.c) La instanciación de la estructura de RTTI del mixin se realiza en el archivo source con la macro `AllocateMixin`:

```
//MyMixin.c
AllocateMixin( MyMixin );
```

Junto con esta macro deben implementarse varias funciones para el mixin: inicialización y finalización de la estructura de RTTI del mixin, constructor, constructor copia y destructor de la instancia del mixin en la clase (estas se llaman automáticamente al crear, copiar o destruir la instancia de la clase) y por último un método `populate` para actualizar la tabla virtual de la clase con los métodos que vayan a implementarse en el mixin mismo:

```
//MyMixin.c
static
void
MyMixin_populate( MyMixin mymixin )
{
    mymixin->myMixinMethod  =  MyMixin_myMixinMethod;
}
```

4.9.5 Dificultades en la codificación

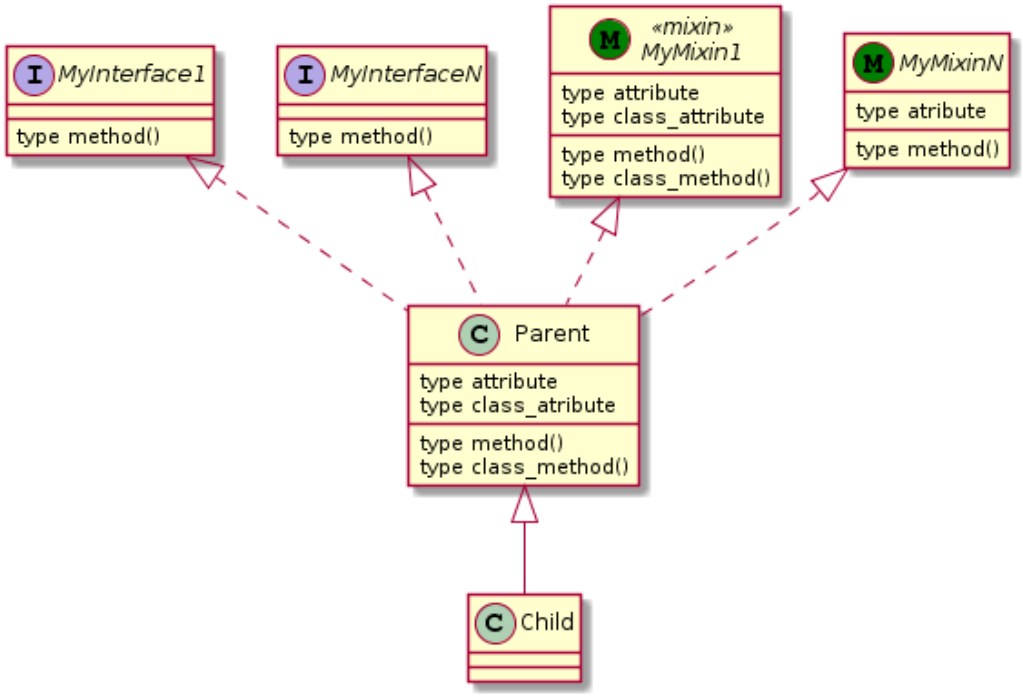
#	Tipo de dificultad	Nombre	Descripción
1	Propensión a errores	Inicialización de métodos polimórficos	Contrario a otros LPOO, a cada referencia de un método debe asignársele su implementación en la función de inicialización de la clase (en el método `initialize()`), si esto se olvida generaremos un error crítico al enviar un mensaje con el método no inicializado. Si se trata de un mixin esto se debe hacer en el método populate.

#	Tipo de dificultad	Nombre	Descripción
2	Propensión a errores y obtención de información	Asignación de métodos redefinidos en la tabla virtual	Para poder asignar un método ya definido por una clase padre se debe conocer exactamente cuál de los padres es el que definió el método por primera vez, y en caso de tratarse de una interfaz se debe conocer cúal es y cuál de los padres la realizó por primera vez, además de que en caso de equivocarnos recibiremos una advertencia por parte del compilador, esto agrega una demora al programador que debe obtener esta información.
3	Repetición de información	Cada método polimórfico nuevo implica 4 referencias al mismo y una redefinición de método implica 3	Si es nuevo se lo referencia al incluirlo en la tabla virtual y tanto si es nuevo como redefinido: en su implementación, y dos veces al asignar la implementación a su referencia en la tabla virtual.
4	Repetición de información	Repetición en la información de la herencia	En las macros `DeclareClass`, `Virtuals` y `AllocateClass`, lo mismo pasa en el método `constructor()` al tener que llamar al mismo método de su padre.
5	Dificultad de aprendizaje	Nuevas macros que aprender	Hay una importante cantidad de macros a aprender y para utilizar correctamente el framework.

#	Tipo de dificultad	Nombre	Descripción
6	Repetición de código	Repetición en cada clase	comparado con otros LPOO, hay mucho código repetido entre una clase y otra tan solo para definir la clase y su herencia, repitiendo varias veces el nombre de la clase como argumento de distintas macros
6	Repetición de información	Duplicación de información al realizar una clase a una interfaz	El hecho de que la clase realiza a una interfaz se debe declarar bajo la macro `Virtuals` y con la macro `InterfaceRegister`.
7	Repetición de información	Triplicación de información de herencia de mixins	Bajo la macro `Virtuals` se declaran los métodos del mixin en la clase, bajo la macro `ClassMembers` se declaran las variables del mixin en la clase y mediante la macro `InterfaceRegister` se lo referencia en la estructura RTTI.

`ooc` ofrece una herramienta para crear clases en base a plantillas para mitigar muchas de estas dificultades (específicamente las dificultades número 3 y 5 y en parte la 4)

4.9.6 Propuesta de expresión en UML



Cualquier clase puede realizar una o varias interfaces, heredar uno o varios mixins, y heredar de hasta una clase. Todas las combinaciones de estas posibilidades son permitidas. Cualquier método en una clase o mixin puede definirse como polimórfico o no (mediante el atributo `isLeaf`), si es de clase o no (mediante el atributo `isStatic`), si es constante o no (mediante el atributo `isConst`), y si es abstracto o no (mediante el atributo `isAbstract`). Las interfaces sólo definen métodos abstractos, pero pueden ser definidos como constantes. Los mixins se representan mediante una interfaz. La misma debe contener atributos o el estereotipo `<< mixin >>`. De esta manera se distinguen de las interfaces. Los atributos de una clase o mixin pueden ser de clase (mediante el atributo `isStatic`) y constantes (mediante el atributo `isConst`). Los atributos de visibilidad son ignorados. Las opciones de visibilidad para `ooc` ya fueron explicadas en la sección Visibilidad.

4.10 Object Oriented C - Simplified (OOC-S) de Laurent Deniau

4.10.1 Introducción

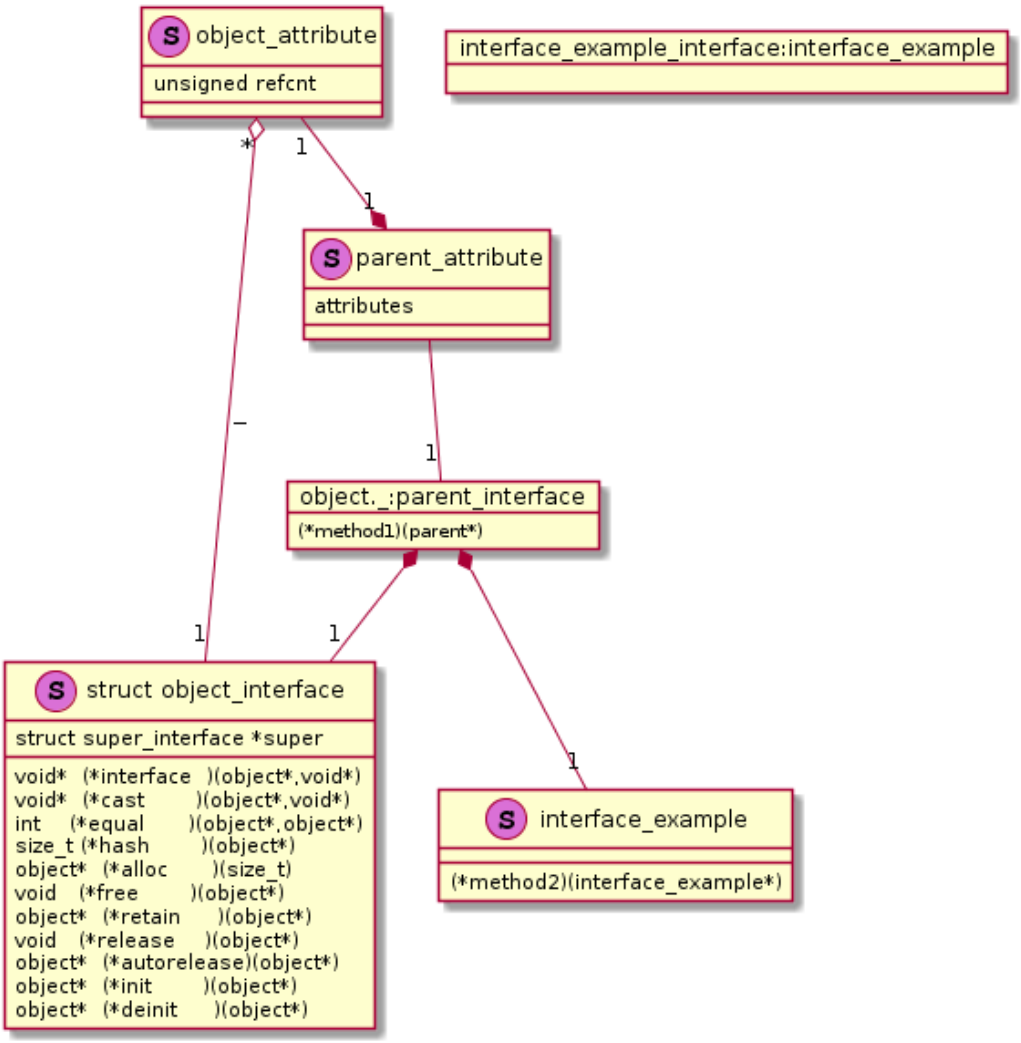
OOC-S busca ser una versión simplificada de OOC-2.0 (Deniau 2007a), un framework en C basado en JAVA y abandonado para el desarrollo de COS. Está escrito en unos pocos cientos de líneas y proporciona algunas pautas de codificación y técnicas de programación muy simples para permitir a los programadores de C escribir programas orientados a objetos casi como en Java. OOC-S proporciona gestión de propiedad (incluida la liberación automática soportada en varios threads con TLS, opción no soportada por la mayoría de los sistemas altamente restringidos por lo que requerirá una modificación para poder ser usado en los mismos). OOC-S requiere un compilador C89. El autor ya nos advierte de sus dificultades de programación: "minimiza el uso de macros (uno por clase) y typedefs (ninguno) y por lo tanto requiere algo de **codificación manual** (inicialización de clase) y algo de **cuidado por parte del programador**. Seguir las plantillas de OOC-S o los ejemplos con todas las opciones de advertencia del compilador habilitadas debería evitar los **errores y olvidos** más comunes. Todavía se pueden construir algunas macros sobre OOC-S para automatizar parte de la codificación". En esta tesis proponemos un generador de código UML en vez o además de macros para sortear estas dificultades.

4.10.2 Conceptos soportados

1. Encapsulamiento
2. Herencia
3. Polimorfismo
4. Interfaces

4.10.3 Modelo de objetos

OOC-S posee un modelo de objetos muy simple. En el siguiente ejemplo mostramos una clase `Parent` que hereda de la clase base `Object` y realiza una interfaz `interface_example`



La clase base de todas las clases es `object`, la misma contiene un único atributo (`refcnt`) que sirve para la gestión de propiedad o de memoria y una referencia a una tabla virtual con varios métodos y a la tabla virtual de la clase padre (para la de `object` es nula). Toda clase definida por el programador (como en el diagrama `parent`) hereda de

object, referencia a una tabla virtual (por ejemplo `parent_interface`) que puede definir nuevos métodos así como incluir interfaces (como `interface_example`), las interfaces son referenciadas mediante una única instancia a la interfaz cuyo nombre es el de la interfaz seguido de `interface` (como `interface_example_interface`). Para obtener la interfaz implementada en un objeto (por ejemplo la interfaz `interface_example` implementada por `parent`) se realiza una llamada al método `interface()` con la referencia a la interfaz. Por ejemplo para obtener la implementación de la interfaz `interface_example` de la clase del objeto referenciado por una variable `obj` se debería ejecutar `obj->i->interface(obj,interface_example_interface)`. Entonces la implementación dada por el usuario del método `interface()` debe, al recibir como segundo argumento la instancia `interface_example_interface`, devolver la instancia de `interface_example` dentro de su tabla virtual.

4.10.4 Codificación

Mostraremos la implementación de la clase `parent` y de la interfaz `interface_example` del ejemplo anterior.

4.10.4.1 Clases

4.10.4.1.1 Visibilidad La visibilidad de las clases en ooc-s es de dos tipos: de declaración (encapsulación fuerte) o de implementación y dependen del archivo de interfaz que incluyamos. La visibilidad de interfaz se obtiene incluyendo el archivo que tiene el nombre de la clase (por ejemplo `clase.h`). La visibilidad de implementación se obtiene incluyendo el archivo que tiene el nombre de la clase seguido por `_p` (por ejemplo `clase_p.h`). Las variables de instancia son accesibles bajo la visibilidad de implementación. Las clases heredadas acceden, en su implementación, a sus clases padres bajo la visibilidad de implementación.

4.10.4.1.2 interfaz (.h)

```
#include "object.h"
#include "interface_example.h"

#define TYPE      struct parent
#define INTERFACE struct parent_interface

#define PARENT_INTERFACE                        \
        OBJECT_INTERFACE                        \
        INTERFACE_EXAMPLE_INTERFACE             \
        void (*method1)(TYPE*);

TYPE {
  INTERFACE *i;
};

INTERFACE {
  struct object_interface *super;
  PARENT_INTERFACE
};

TYPE      *parent_new      (void);
INTERFACE *parent_interface(void);

#undef TYPE
#undef INTERFACE
```

Se puede apreciar, en esta especificación, otra forma de implementar la herencia de estructuras que es mediante un listado de los miembros de la estructura referenciados por una macro. Esto simplifica enormemente la navegación en la estructuras heredadas. En este caso simplifica para el implementador de la clase la inicialización de la tabla virtual y como veremos simplifica también el mapeo a UML. Esto mismo puede hacerse elegantemente bajo ISO C11 con las estructuras anónimas como vimos en la especificación de Ben Klemens, pero esto es realizable incluso bajo ISO C89.

4.10.4.1.3 Interfaz de implementación (_p.h)

Esta interfaz es la que provee la visibilidad de implementación haciendo visibles los atributos de la clase. La siguiente interfaz sigue al ejemplo anterior.

```
#include "parent.h"
#include "object_p.h"

struct parent_attribute {
  struct object_attribute object;
  /* atributos aquí  */
};
```

4.10.4.1.4 Implementación (.c)

Explicaremos el ejemplo por partes. Primero se definen macros de uso común para la clase:

```
#include "parent_p.h"

#define TYPE          struct parent
#define INTERFACE     struct parent_interface
#define SELF        ((struct parent_attribute*)self)
#define AS_SELF(a)  ((struct parent_attribute*)(a) )
#define SUPER       ((struct object          *)self)
#define AS_SUPER(a) ((struct object          *)(a) )
```

Luego se crea una instancia privada de la tabla virtual de la clase, que puede ser inicializada y obtenida mediante la función `parent_interface()`

```
static INTERFACE interface;
```

Luego se define la implementación para parent del método `interfaces()` que es el encargado de devolver las interfaces realizadas por la clase. En el ejemplo si se pide la interfaz `interface_example_interface` se devuelve la posición de esa interfaz dentro de la tabla virtual que corresponde a la

dirección del primer método implementado. En caso de que esta clase no realice la interfaz consultada se reenvía la consulta su padre lo que esto continúa hasta llegar a `object` que retorna con valor `NULL` al no realizar ninguna interfaz.

```
static void*
parent_interfaces(TYPE* self, void *ref_i)
{
  if (ref_i == &interface_example_interface)
    return &self->i->method2;

  /* más interfaces aquí */

  return interface.super->interface(SUPER,ref_i);
}
```

Luego son definidas las implementaciones de los métodos de la clase, en este caso el método `init()` está siendo redefinido.

```
static TYPE*
parent_init(TYPE *self)
{
  interface.super->init(SUPER);
  /* inicialización de atributos  */
  return self;
}

static void
method1(TYPE *self){}

static void
method2(TYPE *self){}
```

Luego se define una función `parent_initialize()` que es la encargada de inicializar la tabla virtual. La tabla virtual es accesible desde afuera a través

de la función `parent_interface()` y en caso de no estar inicializada se llama a `parent_initialize()`. Primero se hace una copia de la tabla virtual del padre, en este caso `object`, en la tabla virtual de la clase, esto permite heredar el comportamiento del padre (la implementación de sus métodos). Luego se realiza una copia de las interfaces por defecto en la posición de la interfaz en la tabla virtual, esto permite heredar los métodos por defecto definidos por la interfaz. Luego a la tabla virtual de la clase se le asignan las implementaciones específicas de los métodos de la clase que pueden sobrescribir los métodos copiados anteriormente, además la referencia a `super`, es decir, a la clase padre es sobrescrita con la correspondiente, en este caso con `object`.

```
static void
parent_initialize(void)
{
  struct object_interface *super_interface =
  ↪  object_interface();

  memcpy(&interface, super_interface, sizeof *super_interface);
  memcpy(&interface.method2, &interface_example_interface,
    sizeof interface_example_interface);

  interface.super      = super_interface;
  interface.interface  = parent_interfaces;
  interface.init       = parent_init;
  interface.method1 = method1;
  interface.method2 = method2;
  /* put parent methods initialization here */
}
```

Por último tenemos las funciones para obtener la tabla virtual y para instanciar un objeto de la clase en memoria dinámica.

```
INTERFACE*
parent_interface(void)
```

```
{
  if (!interface.interface)
    parent_initialize();

  return &interface;
}

TYPE*
parent_new(void)
{
  TYPE *self = parent_interface()->alloc(sizeof *SELF);
  assert(self);

  self->i = &interface;

  return parent_init(self);
}
```

4.10.4.2 Interfaces

4.10.4.2.1 Interfaz (.h

```
#define TYPE      struct interface_example
#define INTERFACE struct interface_example_interface
```

Lo primero que define una interfaz es una macro con un listado de los métodos miembro. Esta macro es incluida en la tabla virtual de las clases que realicen la interfaz.

```
#define INTERFACE_EXAMPLE_INTERFACE                         \
  void (*method2)(TYPE*);                                   \
  /* more interface_example methods here          */        \
  /* TYPE* (*method_example)(TYPE*); */
```

Luego se define un tipo para un objeto que realice la interfaz y una estructura que representa la interfaz, es decir el conjunto de métodos que la componen.

```
 TYPE {
  struct interface *i;
};

INTERFACEINTERFACE
};
```

Por último se da visibilidad pública a la variable que identifica a la interfaz ()`interface_example_interface`), utilizada para obtener la implementación de la misma por parte de cualquier objeto a través del método `interfaces()`. Esta variable, además ,contiene la implementación por defecto de la interfaz (o sea la implementación de sus métodos por defecto).

```
extern INTERFACE interface_example_interface;

#undef TYPE
#undef INTERFACE
```

4.10.4.2.2 Implementación (.c) El archivo de implementación tan sólo consta de la implementación de los métodos por defecto de la interfaz y la inicialización de la variable que identifica a la interfaz y que contiene las implementaciones de métodos por defecto.

```
#include "interface_example.h"

#define TYPE      struct interface_example
#define INTERFACE struct interface_example_interface


/* put INTERFACE default methods implementation here, if any
 ↪  */
```

```
static void method2(TYPE*arg){}

INTERFACE interface_example_interface=
 {method2 }
;
```

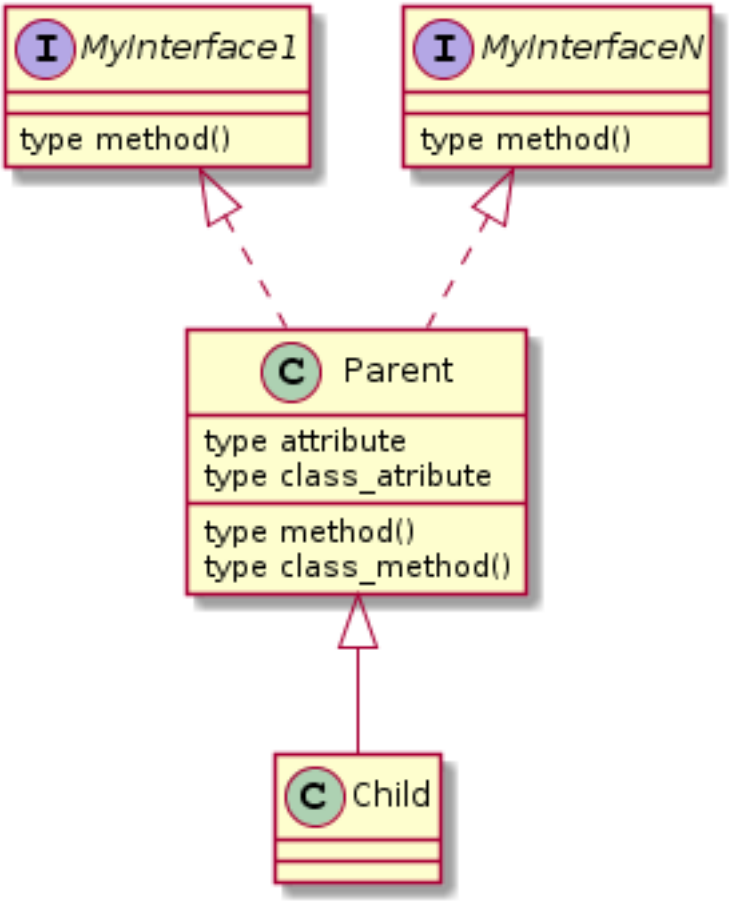
4.10.5 Dificultades en la codificación

#	Tipo de dificultad	Nombre	Descripción
1	Propensión a errores	Inicialización de métodos polimórficos	Contrario a otros LPOO, a cada referencia de un método debe asignársele su implementación en la función de inicialización de la clase (en la función `initialize()`), si esto se olvida generaremos un error crítico al enviar un mensaje con el método no inicializado.
2	Repetición de información	Cada método polimórfico nuevo implica varias referencias al mismo.	
3	Repetición de información	Repetición en la información de la herencia	Varias referencias a la clase padre para poder implementar la herencia.
4	Dificultad de aprendizaje	Convenciones	Si bien esta especificación no posee macros, sí hay que seguir sus convenciones de codificación para estar incluida en la misma.

#	Tipo de dificultad	Nombre	Descripción
5	Repetición de código	repetición en cada clase	comparado con otros LPOO, hay mucho código repetido entre una clase y otra tan solo para definir la clase y su herencia.
6	Repetición de información	Duplicación de información al realizar una clase a una interfaz	El hecho de que la clase realiza a una interfaz implica referenciarla en la definición e inicialización de la tabla virtual del objeto que la realiza, así como en el método interfaces.

Destacamos que este framework no posee la dificultad de tener que conocer qué clase definió por primera vez un método para poder redefinirlo, esto lo logra mediante la redefinición constante de las macro `TYPE`e `INTERFACE`.

4.10.6 Propuesta de expresión en UML



Cualquier clase puede realizar una o varias interfaces y heredar de hasta una clase. Todas las combinaciones de estas posibilidades son permitidas. Cualquier método en una clase puede definirse como polimórfico o no (mediante el atributo `isLeaf`), si es de clase o no (mediante el atributo `isStatic`), si es constante o no (mediante el atributo `isConst`), y si es abstracto o no (mediante el atributo `isAbstract`). Las interfaces pueden definir métodos como abstractos o no (o sea que ofrecen una implementación del método por defecto), como constantes o no y como de clase o no. Los atributos pueden ser de clase (mediante el atributo `isStatic`) y constantes (mediante el atributo `isConst`). Los atributos de visibilidad son ignorados. Las opciones de visibilidad para `ooc-s` ya fueron explicadas en la sección Visibilidad.

4.11 OOC de Axel Tobias Schreiner

4.11.1 Introducción

Este famoso framework fue uno de los primeros trabajos en el área de programación orientada a objetos bajo lenguaje C. Fue desarrollado por un profesor universitario para enseñar orientación a objetos sin la necesidad de aprender un nuevo lenguaje aparte de C. En las palabras del autor, si el alumno ve cómo se hace en un entorno familiar, es mucho más fácil comprender los conceptos y saber qué milagros puede esperar de la técnica (la orientación a objetos) y qué no.
Está basado en punteros `void`. Por esto utiliza bastante programación defensiva para asegurar el uso correcto de los objetos. Esto último significa que verifica que los argumentos pasados por punteros `void` sean realmente objetos. Soporta el reenvío de mensajes (message forwarding) (ver el modelo de objetos) lo que le da mucha flexibilidad y junto con delegaciones le permite programar bajo Duck Typing. Soporta metaclases lo que permite tratar a las clases como objetos y tener absoluto control de la herencia como en lenguajes prototipados (Deniau 2009).

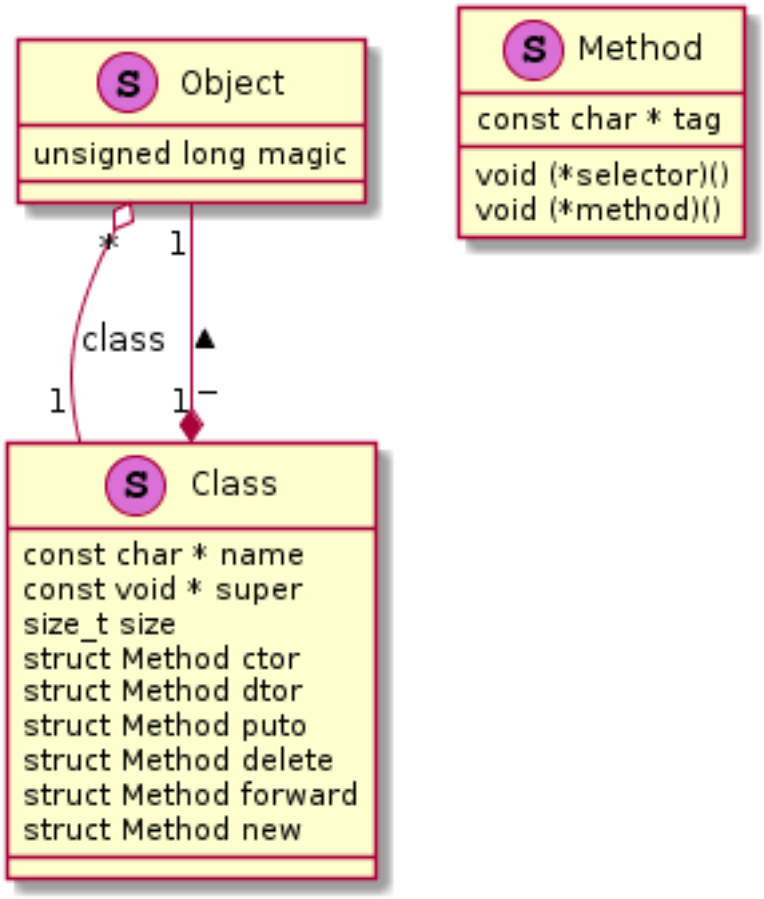
4.11.2 Conceptos soportados

1. Encapsulamiento
2. Herencia
3. Polimorfismo
4. Reenvío de mensajes (message forwarding) / Delegados (Delegates)
5. Metaclases
6. Excepciones

4.11.3 Modelo de objetos

El modelo de objetos de ooc es bastante sencillo para la cantidad de conceptos que soporta. Mantiene una jerarquía distinta para las clases y las metaclases.

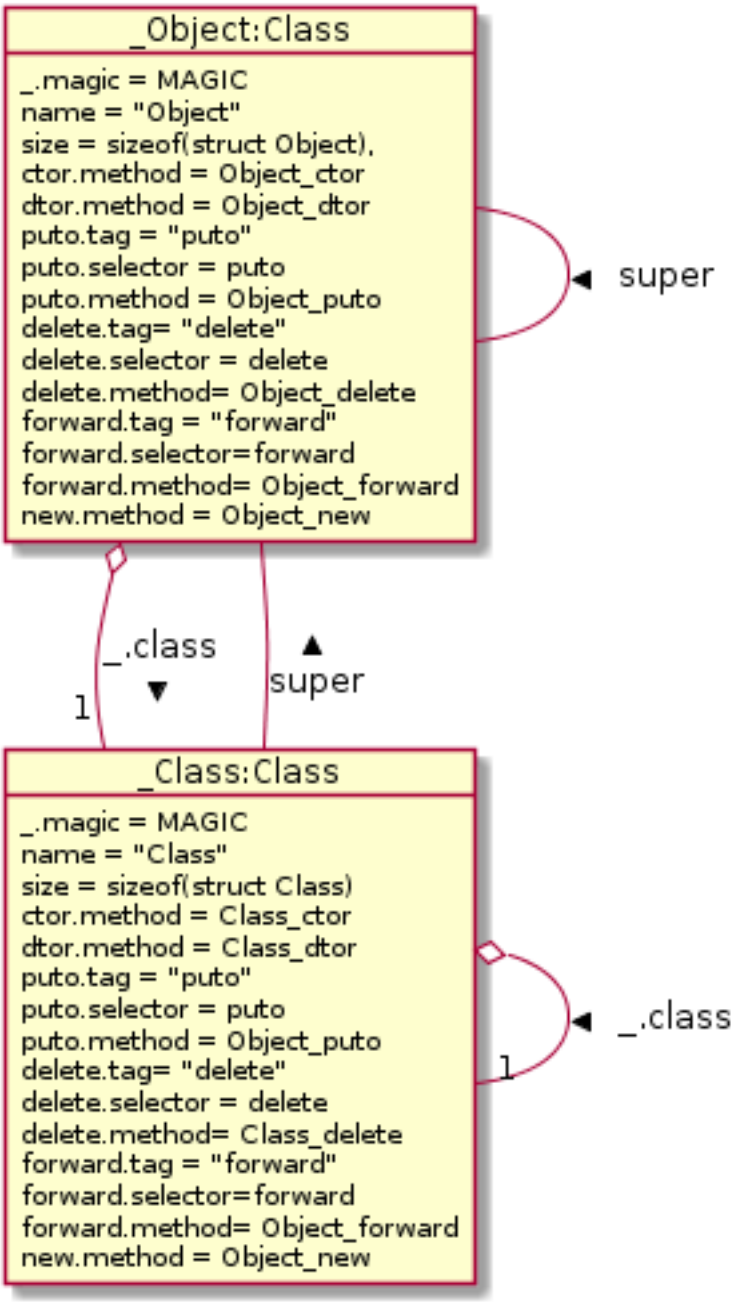
El siguiente diagrama representa las estructuras que dan soporte al modelo de objetos.



Una instancia de la estructura `Object` es un objeto, una instancia de la estructura `Class` es una clase. Todas las clases heredan de la clase `Object`. La misma está compuesta de un número llamado `magic`, que sirve para saber si cierta variable es un objeto y tiene el mismo valor para todos los objetos. Esto se utiliza para la programación defensiva y asegurarse de que un objeto pasado por argumento en un puntero `void` sea realmente un objeto. Además contiene a su clase que se referencia con la variable `class` y es una instancia de la estructura `Class`. La referencia a su clase contiene su nombre, una referencia a su clase padre, su tamaño y los métodos polimórficos de instancia. Los métodos polimórficos de instancia por defecto son su constructor (`ctor()`), su destructor (`dtor()`), un serializador de la clase (`puto()`), un instanciador (`new()`) y desinstanciador (`delete()`) y

un método que se encarga del reenvío de mensajes (`forward()`) como se explicará. Una clase a la vez hereda de `Object`, por lo que la convierte en un objeto que tiene su propia referencia a su clase. Esta última clase (la clase de una clase) se llama una metaclase y contiene los mismos métodos polimórficos por defecto que ya nombramos. Esto significa, por ejemplo, que a través de una metaclase puede instanciarse una nueva clase mediante el método `new()`. Este método permite redefinir los métodos que se deseen para la nueva clase. Los métodos definidos en la metaclase son métodos de clase.

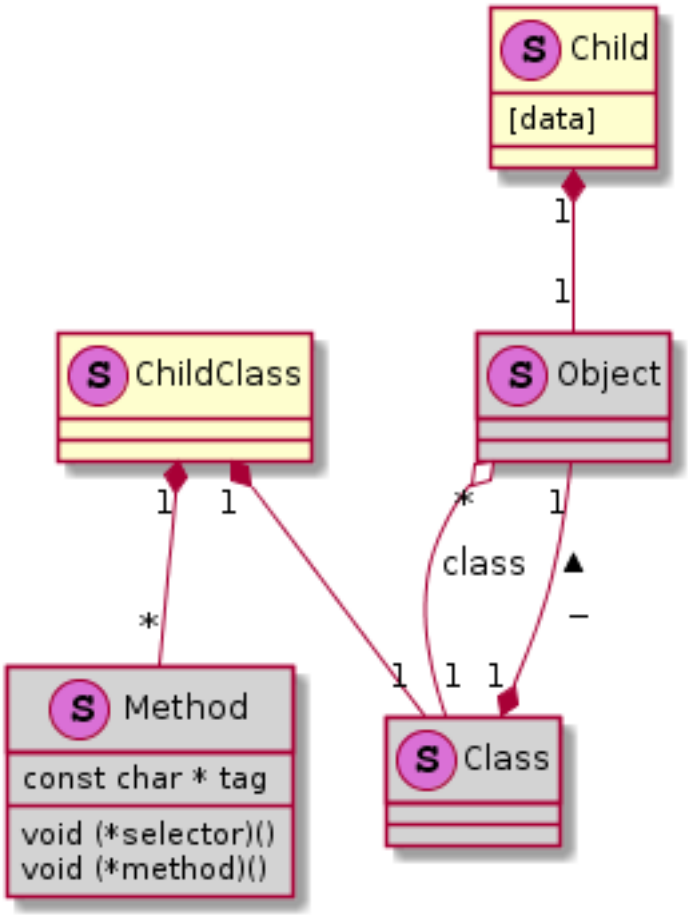
El siguiente diagrama representa a la instancia que representa a la clase `Object` y a la que representa a la metaclase `Class`



En estas estructuras básicas es interesante hacer notar que la clase `Object` desciende de sí misma y la clase de la metaclase `Class` es `Class` (lo que significaría su meta-metaclase) y así continúan recursivamente. Esto es lo que se llama un modelo de objetos uniforme, donde tanto las clases como los objetos son objetos con el mismo uso (Deniau 2009). Cada método en una clase se define con la estructura `Method`, la misma contiene un tag (bajo el

nombre `tag`), una implementación (`method`) y un selector (`selector`). El tag permite referenciar a un método mediante una cadena de caracteres. Esto permite crear módulos desacoplados pero que pueden interactuar entre sí mediante convención de nombres. Esto se logra a través de un método no polimórfico llamado `respondsTo()` que recibe como argumentos el objeto a interrogar si posee un método con cierto tag y el tag mismo. La función devuelta por `respondsTo()` es la del selector y no la de la implementación, por eso se incluye al selector en esta estructura ya que, de lo contrario, se lo puede llamar directamente por su prototipo declarado en el archivo de interfaz. Los objetos que poseen métodos que se ajustan a las convenciones de otros objetos, permitiéndoles a estos últimos utilizarlos, se llaman delegados (delegates en inglés). El selector (o despachador, dispatcher en inglés) es el encargado de seleccionar la implementación correcta del método al cual representa y ejecutarlo. Para eso recibe como argumento el objeto que se supone implementa el método en cuestión. En caso de que no lo implemente se llama al método `forward()` del objeto con la información del mensaje que se envió al método y desde ahí el objeto puede decidir reenviar el mensaje a otro objeto o procesarlo de otra forma (por ejemplo imprimiendo un error y terminando el programa). Esta capacidad de reenviar un mensaje hacia otro método se llama message forwarding y provee una excepcional flexibilidad al programador. La misma puede suplir la necesidad de la herencia múltiple mediante la composición con otras clases y reenviando los mensajes a las clases que implementan dichos mensajes.

Tanto las clases como las metaclases pueden heredarse. Heredar una metaclase permite definir nuevos métodos y permitir instanciar nuevas clases que implementen dichos métodos. El siguiente diagrama representa las estructuras de una clase `Child` que hereda de `Object` y que define también una nueva metaclase que hereda de `Class`.



`Child` incluye a `Object` como primer elemento lo que convierte a un `Child` en un `Object`, la estructura `Child` puede contener más atributos de instancia (`data` en el diagrama). Para definir nuevos métodos, se incluyen en la estructura de la que se instancia que representa a la clase `Child`, `ChildClass`, que no puede incluir atributos de clase (esto se debe a que `respondsTo()` espera que haya solo métodos). La metaclase seguirá siendo una instancia de la estructura `Class` pero cambiará principalmente su constructor para construir instancias de la estructura `ChildClass` (una de ellas la clase de `Child` misma). Si un nuevo objeto desea heredar de `Child` pero no define nuevos métodos de instancia o métodos y atributos de clase, entonces heredará de la estructura `Child` pero no precisa heredar de la estructura `ChildClass`, ya que la metaclase de `Child` permite instanciar una nueva clase con las implementaciones de los métodos definidos para la nueva clase (al igual que sus tags).

4.11.4 Codificación

El autor del framework reconoce las dificultades de implementar clases bajo este framework y provee un preprocesador escrito en AWK para facilitar su codificación. Luego analizaremos el uso de preprocesadores distintos al provisto por C para sortear estas dificultades. En esta tesis buscamos presentar generadores de código desde UML para resolver dichas dificultades. Lo que nos interesa aquí es mostrar qué partes del código C conllevan esa dificultad para evaluar más adelante si un generador de código desde UML podría factiblemente resolverlas.

4.11.4.1 Visibilidad

Las posibilidades de visibilidad de los miembros de clase (atributos y métodos) son iguales a los del framework `ooc` de Tibor Miseta. Los métodos privados se declaran en un archivo con el nombre de la clase y extensión `.c`, los métodos públicos con extensión `.h` y los métodos de implementación con extensión `.r`. Todos los atributos tienen visibilidad de implementación. Para el autor, la visibilidad de implementación ,en principio, es sólo para las clases derivadas (y obviamente también para la clase implementadora) por lo que sería equivalente a la visibilidad protegida para otros LPOO. Para los sistemas altamente restringidos, sin embargo, resulta importante poder instanciar objetos en variables automáticas o estáticas por lo que implicaría incluir este archivo en más fuentes que solo las de esas clases.

4.11.4.2 Archivo de interfaz de implementación (.r)

```
//Child.r
# include "Object.r"

struct Child { const struct Object _;
    /*Child data*/
};
```

```
struct ChildClass { const struct Class _;
    struct Method myMethod;
    /*more methods here*/
};

struct Object * super_myMethod (const void * _class,
                        void * _self, void * anObject);
```

En este archivo se definen las estructuras de instancia de clase (en este caso las instancias son objetos de tipo `Child`) y de instancias de metaclase (clases de tipo `ChildClass`). Al comienzo se incluye el archivo de interfaz de implementación de la clase padre (`Object`). En la estructura `Child` se declaran variables de instancia. En la estructura `ChildClass`, los métodos de polimórficos. El selector de los métodos polimórficos declarados (en el ejemplo `myMethod`) se declara en el archivo `.h`. El método `super_myMethod()` se utiliza para llamar a la implementación del método `myMethod` en la clase padre.

4.11.4.3 Archivo de implementación (.c)

```
//Child.c
int Child_myMethod (void * _self, void * anObject) {
    /*...*/
}

int myMethod (void * _self, void * anObject) {
    int result;
    const struct ChildClass * class =
                    (const void *) classOf(_self);

    if (isOf(class, ChildClass())
                && class -> myMethod.method) {
        cast(Object(), anObject);
```

```
        result =
            ((struct Object * (*) ()) class -> myMethod.method)
                (_self, anObject);
    } else
        forward(_self, & result, (Method) add, "add",
                                                _self, anObject);
    return result;
}

int super_myMethod (const void * _class, void * _self,
                    void * anObject) {
    const struct ChildClass * superclass =
                                        cast(ChildClass(),
↪   super(_class));

    cast(Object(), anObject);

    assert(superclass -> myMethod.method);
    return ((struct Object * (*) ()) superclass ->
↪   myMethod.method)
                    (_self, anObject);
}

static void Child_forward (const void * _self, void * result,
                           Method selector,
                           const char * name,
                           va_list * app) {
    const struct Child * self = cast(Child(), _self);

    if (selector == (Method) otherMethod)
    {
        /* process otherMethod message */
    }
    else
```

```
        super_forward(Child(), _self, result, selector, name,
↪   app);
}

static void * ChildClass_ctor (void * _self, va_list * app) {
    struct ChildClass * self = super_ctor(ChildClass(), _self,
    ↪   app);
    Method selector;
    va_list ap;
    va_copy(ap,*app);

    while ((selector = va_arg(ap, Method)))
    {   const char * tag = va_arg(ap, const char *);
        Method method = va_arg(ap, Method);

        if (selector == (Method) myMethod)
        {   if (tag)
                self -> myMethod.tag = tag,
                self -> myMethod.selector = selector;
            self -> myMethod.method = method;
            continue;
        }
        /*other methods*/
    }
    return self;
}

static const void * _ChildClass;

const void * const ChildClass (void) {
    return _ChildClass ? _ChildClass :
        (_ChildClass = new(Class(),
            "ChildClass", Class(), sizeof(struct ChildClass),
            ctor, "", ChildClass_ctor,
            (void *) 0));
```

```
}

static const void * _Child;

const void * const Child (void) {
    return _Child ? _Child :
        (_Child = new(ChildClass(),
            "Child", Object(), sizeof(struct Child),
            myMethod, "myMethod", Child_myMethod,
            forward, "forward", Child_forward,
            /* other methods here*/
            (void *) 0));
}
```

`Child_myMethod()` es la implementación de `myMethod` para `Child`, sólo se declara en este archivo por lo que tiene visibilidad privada . La función `myMethod()` es el selector del método, se declara en el archivo de interfaz por lo que tiene visibilidad pública, realiza las verificaciones de tipos de los argumentos y verifica que el objeto al cual se envía el mensaje sea del tipo que definió ese método (en este caso `Child`), si no se reenvía el mensaje al método `forward()` del objeto. `super_myMethod()` ya fue analizado, también verifica que el tipo del objeto al cual se le envía el mensaje sea `ChildClass`. Se muestra en el ejemplo una implementación del método `forward()` para procesar una llamada a un objeto `Child` con el método `otherMethod`, el mismo podría reenviarse a un objeto que comprenda ese mensaje.
La función `Child()` retorna a la clase `Child`, si es la primera vez que se la llama, se la instancia en ese momento. Al constructor se le indican las implementaciones de los métodos y sus tags que debe contener la clase `Child`, en este caso las ternas selector, tag, implementación para `myMethod` y para `forward`. Cómo se procesan estos argumentos y se asignan a la clase se lo puede ver en la función `ChildClass_ctor()`. La función `ChildClass()` devuelve una instancia de la metaclase `ChildClass`, que desciende de `Class`, y tiene sigue la misma lógica de argumentos para se construcción.

4.11.5 Dificultades en la codificación

#	Tipo de dificultad	Nombre	Descripción
1	Propensión a errores	Inicialización de métodos polimórficos	Contrario a otros LPOO, a cada referencia de un método debe asignársele su implementación a través del método de inicialización de la clase, si esto se olvida generaremos un error crítico al enviar un mensaje con el método no inicializado.
2	Repetición de información y obtención de información	Cada método polimórfico implica varias referencias al mismo	Si es la primera vez que se define el método: en el selector, el selector "superclass", su asignación en el constructor de la metaclase, y tanto sea una definición como una redefinición: su implementación y como argumento en el constructor de clase. Es necesario obtener la información de sí es la primera vez que se define el método o no.
3	Repetición de información	Repetición en la información de la herencia tanto para clases como para metaclases	Además de indicarse esto mediante la herencia de estructuras en el archivo de interfaz de implementación se lo indica como argumento del constructor de la clase o de la metaclase.

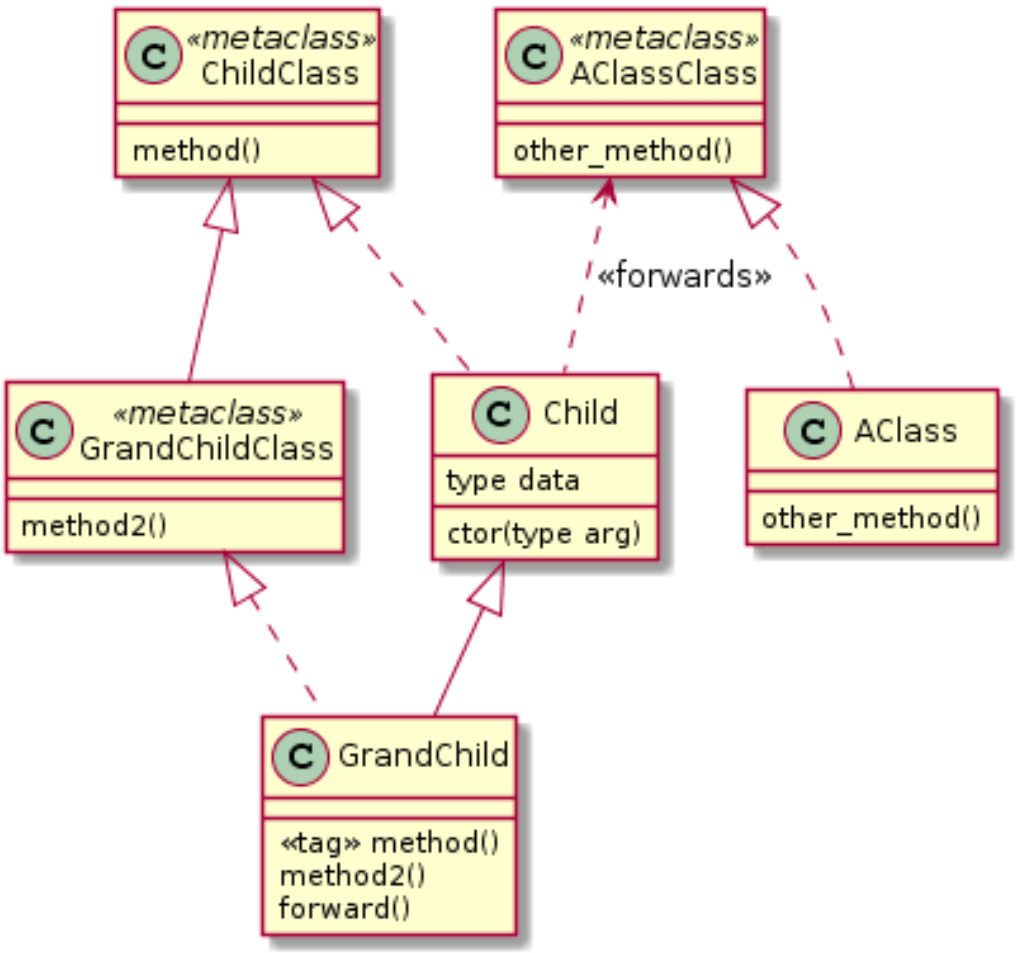
#	Tipo de dificultad	Nombre	Descripción
4	Repetición de código	Nombre de la clase en cada implementación de método o en un método no polimórfico	Para no colisionar con los nombres utilizados por otras clases, o con el selector, todas los métodos antedichos llevan por delante el nombre de la clase.
5	Repetición de código y exigencia de codificación	Repetición en los métodos selectores y constructores de metaclase	Los métodos selectores y constructores de metaclase son prácticamente iguales o siguen la misma lógica para toda clase y método polimórfico. Además de eso implican una gran cantidad de código.
6	Exigencia de codificación	La programación defensiva implica codificar gran cantidad de verificaciones (mediante la función `cast()`)	Prácticamente no existe un método donde no haya que realizar alguna verificación de los argumentos que se le pasan por `void*`, sólo las implementaciones de métodos cuyos selectores ya hicieron las verificaciones pertinentes pueden obviarlas.
7	Propensión a errores y exigencia de codificación	El reenvío de mensajes implica un manejo complicado de los argumentos	Para poder enviar cualquier mensaje al método `forward()` el mismo utiliza argumentos variables que luego deben recuperarse mediante `va_arg()`. Fácilmente puede equivocarse el orden en que están guardados tales argumentos llevando a un error en el código.

#	Tipo de dificultad	Nombre	Descripción
8	propensión a errores y exigencia de codificación	Los constructores de instancia reciben sus argumentos en una lista de argumentos variable	Fácilmente puede equivocarse el orden en que se definieron tales argumentos llevando a un error en el código. Además se debe incluir una llamada a `va_arg()` por cada uno.

4.11.6 Propuesta de expresión en UML

Vemos dos formas convenientes de representar este framework en UML, uno con metaclases explícitas y otra con implicitas.

4.11.6.1 Propuesta de expresión con metaclases explícitas

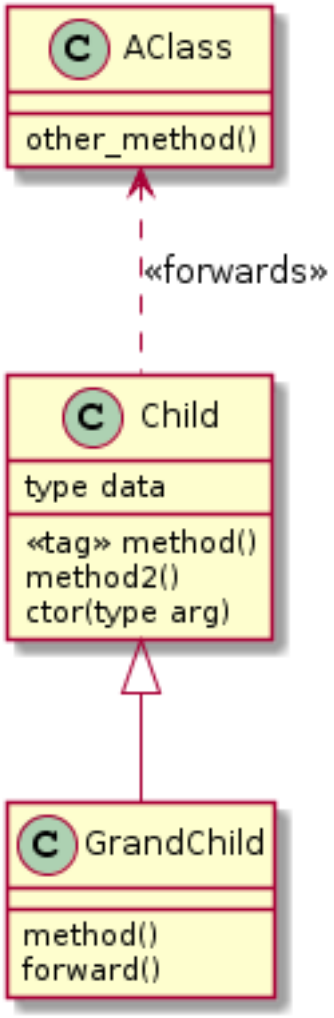


Se puede apreciar la representación tanto de clases como de las metaclases de las cuales se instancian. Las metaclases poseen el estereotipo `metaclass`. La relación entre una clase una metaclase se indica con la relación `realize` que se utiliza para relacionar una clase con la interfaz que realiza. Una clase sólo puede ser instanciada desde una metaclase. Una metaclase hereda de otras metaclases (o de la metaclase `Class` si no se le especifica herencia) y una clase hereda de otra clase (o de `Object` si no se le especifica herencia). Si una clase no tiene representada una metaclase de la cual se instancia entonces es instancia de la metaclase `Class`.

Los métodos en una metaclase describen los métodos polimórficos de

instancia de las clases que son instanciadas de dicha metaclase. Una clase que provea una implementación para alguno de dichos métodos debe incluir al método como una operación propia. Una clase que no posea una implementación ni propia ni heredada de alguno de los métodos de la metaclase de la cual se instancia, es una clase base (en el ejemplo la clase `Child` no implementa el método `method()`) y alguna clase hija puede proveer una implementación para los mismos (en el ejemplo la clase `GrandChild` define una implementación de `method()`). Los métodos `ctor()`, `dtor()`, `puto()`, `new()`, `delete()` y `forward()` definidos en una clase son considerados implementaciones de los métodos definidos por la metaclase `Class`. A las implementaciones de métodos polimórficos se les puede aplicar el estereotipo `<<tag>>`. El mismo contiene un atributo `nombre` que lo representa (el mismo lo referencia para obtenerlo a través del método `respondsTo()`). Si el nombre estuviese vacío se le asignaría el nombre del método. Las variables de instancia (`data`) pueden ser de clase (lo que significa que se instancian en el archivo de implementación de la clase (.c)) y constantes. Los métodos no polimórficos se implementan en el archivo de implementación de la clase (.c) y si tienen visibilidad pública se declaran en el archivo de interfaz de la clase. De acuerdo a `ooc` los constructores reciben sus argumentos en una lista de argumentos variables, sin embargo su representación es con una lista de argumentos fija. Para especificar la capacidad de procesar un mensaje definido por una metaclase (que no pertenece a la jerarquía de metaclases de la metaclase que se realiza) a través del método `forward()` se utiliza un estereotipo que extiende la relación `usage` y se llama `<<forwards>>`. El final de dicha relación (el proveedor, supplier en inglés) es el método mismo y no la metaclase que lo contiene. Se escogió la relación `usage` ya que según la especificación no define el uso que el cliente da al proveedor (OMG 2017), pero mediante el estereotipo sí lo especificamos. También se puede declarar el método `forward()` dentro de una clase para implementarlo (como es el caso en `GrandChild`).

4.11.6.2 Propuesta de expresión con metaclases implícitas



Las clases que definan nuevos métodos polimórficos (es decir, que una clase padre no lo haya ya definido), generan metaclases con el nombre de la clase seguido por `Class`, además implican selectores y selectores de clase padre (super class selector). En caso de no poseer métodos polimórficos nuevos, tanto por esa clase como por las clases padre, entonces la metaclase de dicha clase es `Class`. Los métodos `ctor()`, `dtor()`, `puto()`, `new()`, `delete()` y `forward()` definidos en una clase no se consideran métodos nuevos ya que ya son definidos para `Object`. A los métodos polimórficos se les puede aplicar el estereotipo `<<tag>>`. El mismo contiene un atributo `nombre` que

lo representa (el mismo lo referencia para obtenerlo a través del método `respondsTo()`). Si el nombre estuviese vacío se le asignaría el nombre del método. Las variables de instancia (`data`) pueden ser de clase (lo que significa que se instancian en el archivo de implementación de la clase (.c)) y constantes. Los métodos no polimórficos se implementan en el archivo de implementación de la clase (.c) y si tienen visibilidad pública se declaran en el archivo de interfaz de la clase. De acuerdo a `ooc` los constructores reciben sus argumentos en una lista de argumentos variables, sin embargo su representación es con una lista de argumentos fija. Para especificar la capacidad de procesar un mensaje definida por otra clase a través del método `forward()` se utiliza un estereotipo que extiende la relación `usage` y se llama `<<forwards>>`. El final de dicha relación (el proveedor, supplier en inglés) es el método mismo y no la clase que lo contiene. Se escogió la relación `usage` ya que según la especificación no define el uso que el cliente da al proveedor (OMG 2017), pero mediante el estereotipo sí lo especificamos. También se puede declarar el método `forward()` dentro de una clase para implementarlo (como es el caso en `GrandChild`).

4.12 GObject de GLib

4.12.1 Introducción

Glib es un paquete de bibliotecas C de código abierto y para uso de propósito general. Dentro del paquete Glib se encuentra la biblioteca GObject o GLib Object System, la misma ofrece un sistema de tipos y objetos para C que da soporte para conceptos de la POO. Hoy en día, GObject se utiliza en una variedad de proyectos, incluyendo GIMP, GNOME, Evolution y Gstreamer. Todos esos proyectos demuestran que la programación OO en C es ciertamente posible incluso fuera de los sistemas embebidos (Hendrickx 2004). Glib es portable a distintos sistemas operativos pero no es simplemente portable a un sistema embebido sin sistema operativo.

4.12.2 Conceptos soportados

1. Encapsulamiento
2. Herencia
3. Polimorfismo
4. Interfaces (sin herencia de interfaces)
5. Propiedades
6. Señales (sistema de notificaciones y mensajes de bajo acoplamiento entre clases)
7. Programación clave-valor

4.12.3 Modelo de objetos

Un análisis detallado del modelo de objetos de GObject queda fuera del alcance de esta tesis, aunque no difiere en esencia con la de otros frameworks ya vistos. Todas las clases heredan de una clase común llamada GObject, la misma a su vez hereda de un tipo (no un objeto) llamado GTypeInstance que es la base para otros tipos comunes. La estructura de RTTI de las clases se crea en RAM y en tiempo de ejecución y posibilita agregar manejadores de señales y propiedades en tiempo de ejecución. Las interfaces heredan de GTypeInterface y pueden poseer implementaciones por defecto de sus métodos. Las señales y propiedades son comparables a los delegados en OOC de Schreiner, una forma de acceder a métodos (asociados por una clase con cierto texto) mediante un texto en lugar de mediante una interfaz.

4.12.4 Codificación

En el siguiente ejemplo de codificación la clase `Child` que hereda de `GObject` posee un atributo público (`publicAttr`) y uno privado (`privateAttr`), una propiedad para su atributo público, un método polimórfico y una señal.

4.12.4.1 Interfaz de clases (.h)

```
#include <glib-object.h>

typedef struct _ChildClass ChildClass;
typedef struct _Child      Child;
```

Primero se declara la tabla virtual de la clase, la misma debe comenzar con un `_`, seguir con el nombre de la clase y terminar con la palabra `Class`. El primer miembro es la tabla virtual de la clase de la cual hereda, en este caso GObject, el nombre de este miembro es a elección aunque el nombre `parent` es lo usual.

```
struct _ChildClass {
    GObjectClass parent;
    void (*method) (Child *self);
    void (*signal1Process) (Child *self);
};
```

Luego se declara la estructura de instanciación de la clase, la misma debe comenzar con un `_`, y terminar con el nombre de la clase. El primer miembro es la estructura de instanciación de la clase de la cual hereda, en este caso GObject, el nombre de este miembro es a elección aunque el nombre `parent` es lo usual. Los atributos públicos son declarados a continuación. Si la clase es final, o sea que no es heredable, esta estructura se define en el archivo de implementación.

```
struct _Child {
    GObject  parent;
    gint     publicAttr;
};
```

Las siguientes seis macros se deben declarar para la clase.

```
#define TYPE_CHILD            (child_get_type             ())
#define CHILD(obj)            (G_TYPE_CHECK_INSTANCE_CAST
↪  ((obj), \
                TYPE_CHILD, Child))
#define CHILD_CLASS(cls)      (G_TYPE_CHECK_CLASS_CAST
↪  ((cls), \
                TYPE_CHILD, ChildClass))
#define IS_CHILD(obj)         (G_TYPE_CHECK_INSTANCE_TYPE
↪  ((obj),\
                TYPE_CHILD))
#define IS_CHILD_CLASS(cls)   (G_TYPE_CHECK_CLASS_TYPE
↪  ((cls),\
                TYPE_CHILD))
#define CHILD_GET_CLASS(obj) (G_TYPE_INSTANCE_GET_CLASS
↪  ((obj),\
                TYPE_CHILD, ChildClass))
```

La función `child_get_type()` devuelve una referencia a la clase registrada en el sistema de tipos de glib.

```
GType  child_get_type  (void);
```

Por último se declaran los selectores de los métodos polimórficos.

```
void   child_method     (Child *self);
```

4.12.4.2 implementación de clases (.c)

```
#include <glib-object.h>
#include "child.h"

#include "my_interface1.h"
```

Definiciones para referenciar propiedades.

```
enum {
    PROP_0,
    PROP_ATTR
};
```

Definiciones para referenciar señales:

```
enum {
  SIGNAL1,
  LAST_SIGNAL
};
```

Estructura para atributos privados:

```
typedef struct _ChildPrivate ChildPrivate;

struct _ChildPrivate {
    int privateAttr;
};

static guint child_signals[LAST_SIGNAL] = {0};
```

Selector de `method()`:

```
void child_method (Child *self) {
    CHILD_GET_CLASS(self)->method(self);
}
```

Implementaciones de métodos y señales. `method1()` es definido por `MyInterface1`:

```
static void child_method_impl (Child *self) {
    /* ...  */
}
```

```
static void child_method1_impl (Child *self) {
    /* ...  */
}

static void child_signal1_process (Child *self) {
    /* ...  */
}
```

Métodos get y set para dar soporte a las propiedades definidas por la clase:

```
static void child_get_property (GObject    *obj,
                                guint       prop_id,
                                GValue     *value,
                                GParamSpec *pspec) {
    Child *child = CHILD(obj);

    switch (prop_id) {
    case PROP_ATTR:
        g_value_set_int(value, child->publicAttr);
        break;
    default:
        G_OBJECT_WARN_INVALID_PROPERTY_ID(obj, prop_id, pspec);
        break;
    }
}

static void child_set_property (GObject      *obj,
                                guint         prop_id,
                                const GValue *value,
                                GParamSpec   *pspec) {
    Child *child = CHILD(obj);

    switch (prop_id) {
    case PROP_ATTR: {
```

```
            gint new_attr = g_value_get_int(value);
            child->publicAttr = new_attr;
            break;
        }
        default:
            G_OBJECT_WARN_INVALID_PROPERTY_ID(obj, prop_id, pspec);
            break;
        }
}
```

Inicialización de la clase, de la clase, se asocia a la clase la estructura de atributos privados, las propiedades y señales (estas últimas pueden asociarse con la clase incluso luego de su inicialización), se inicializan los métodos de la tabla virtual con las implementaciones de los métodos y las señales.

```
static void child_class_init (ChildClass *cls) {
    GObjectClass *g_object_class = G_OBJECT_CLASS(cls);
    GParamSpec *attr_param;

    g_type_class_add_private (cls, sizeof (ChildPrivate));


    g_object_class->get_property = child_get_property;
    g_object_class->set_property = child_set_property;

    cls->signal1Process = child_signal1_process;
    cls->method = child_method_impl;

    attr_param = g_param_spec_int(
        "attr", "attr", "attribute of child",
        INT_MIN, INT_MAX,
        0,
        G_PARAM_READWRITE);

    g_object_class_install_property(
```

```
        g_object_class,
        PROP_ATTR,
        attr_param);


    child_signals[SIGNAL1] = g_signal_new(
       "signal1",                                /* signal_name */
        TYPE_CHILD,                               /* itype */
        G_SIGNAL_RUN_LAST | G_SIGNAL_DETAILED, /* signal_flags
↪   */
        G_STRUCT_OFFSET(ChildClass,
↪   signal1Process),/*class_offset*/
        NULL,                                     /* accumulator */
        NULL,                                      /* accu_data */
        g_cclosure_marshal_VOID__VOID,          /* c_marshaller
↪   */
        G_TYPE_NONE,                              /* return_type */
        0);                                        /* n_params */
}
```

Inicialización de la interfaz en la tabla virtual:

```
static void child_init_my_interface1(MyInterface1Iface* iface,
    gpointer iface_data) {
    iface->method1 =
    (void (*)(MyInterface1 *instance))child_method1_impl;
}
```

Registro de la clase y asociación con sus interfaces. Luego del registro se obtiene la referencia a la clase que es siempre devuelta por la función `get_type()`:

```
GType child_get_type (void) {
    static GType child_type = 0;
```

```
    if (!child_type) {
        static const GTypeInfo child_info = {
           sizeof (ChildClass),               /* class_size */
            NULL,                              /* base_init */
           NULL,                             /* base_finalize */
           (GClassInitFunc) child_class_init,  /* class_init */
            NULL,                              /*
↪   class_finalize */
            NULL,                             /* class_data */
            sizeof (Child),                   /* instance_size
             ↪  */
            0,                               /* n_preallocs */
           NULL,                             /* instance_init */
            NULL                              /* value_table */
        };

        child_type = g_type_register_static(
            G_TYPE_OBJECT, /* parent_type */
            "Child",       /* type_name */
            &child_info,   /* info */
            0);            /* flags */

        /* add interface */
        GInterfaceInfo interface_info_my_interface1 = {
            /* interface_init */
            (GInterfaceInitFunc)child_init_my_interface1,
            NULL,   /* interface_finalize */
            NULL,   /* interface_data */
        };
        g_type_add_interface_static(child_type,
            TYPE_MYINTERFACE1, &interface_info_my_interface1);

    }
```

```
    return child_type;
}
```

4.12.4.3 Interfaz de una interfaz (.h)

```
typedef struct _MyInterface1      MyInterface1;
typedef struct _MyInterface1Iface MyInterface1Iface;
```

Se define la estructura de la interfaz que contiene los métodos de la misma y que hereda de `GTypeInterface`.

```
typedef struct _MyInterface1Iface {
    GTypeInterface parent;
    void (*method1)(MyInterface1 *instance);
} MyInterface1Iface;
```

Los siguientes cuatro macros deben definirse para cada interfaz.

```
#define TYPE_MYINTERFACE1           (my_interface1_get_type())
#define MYINTERFACE1(obj)           (G_TYPE_CHECK_INSTANCE_CAST
↪  \
                    ((obj), TYPE_MYINTERFACE1,
↪  MyInterface1Iface))
#define IS_MYINTERFACE1(obj)        (G_TYPE_CHECK_INSTANCE_TYPE
↪  \
                    ((obj), TYPE_MYINTERFACE1))
#define MYINTERFACE1_GET_IFACE(obj)
↪  (G_TYPE_INSTANCE_GET_INTERFACE \
                    ((obj), TYPE_MYINTERFACE1,
↪  MyInterface1Iface))
```

`get_type()` devuelve una referencia única a la interfaz reconocida por el sistema de tipos de glib:

```
GType     my_interface1_get_type                  (void);
```

Selectores de los métodos:

```
void      my_interface1_method1                   (MyInterface1
↪  *self);
```

4.12.4.4 Implementación de una interfaz (.c)

Implementación de los selectores:

```
void my_interface1_method1 (MyInterface1 *self) {
    MYINTERFACE1_GET_IFACE(self)->method1(self);
}
```

Métodos por defecto de los métodos definidos por la interfaz:

```
void my_interface1_impl_method1 (MyInterface1 *self) {
    /* ...  */
}
```

Mediante la inicialización de la interfaz se asignan las implementaciones por defecto, además podrían tomarse recursos para la misma:

```
static void my_interface1_iface_init (MyInterface1Iface *iface)
↪  {
    iface->method1 = my_interface1_impl_method1;
}
```

Registro de la interfaz en el sistema de tipos de glib. Luego del registro `get_type()` devuelve siempre la misma referencia:

```
GType my_interface1_get_type (void) {
    static GType type = 0;
    if (type) return type;

    static const GTypeInfo info = {
        sizeof (MyInterface1Iface),
        NULL,
        NULL,
        (GClassInitFunc) my_interface1_iface_init
    };

    type = g_type_register_static(
        G_TYPE_INTERFACE,
        "MyInterface1",
        &info,
        0);

    return type;
}
```

x

4.12.5 Dificultades en la codificación

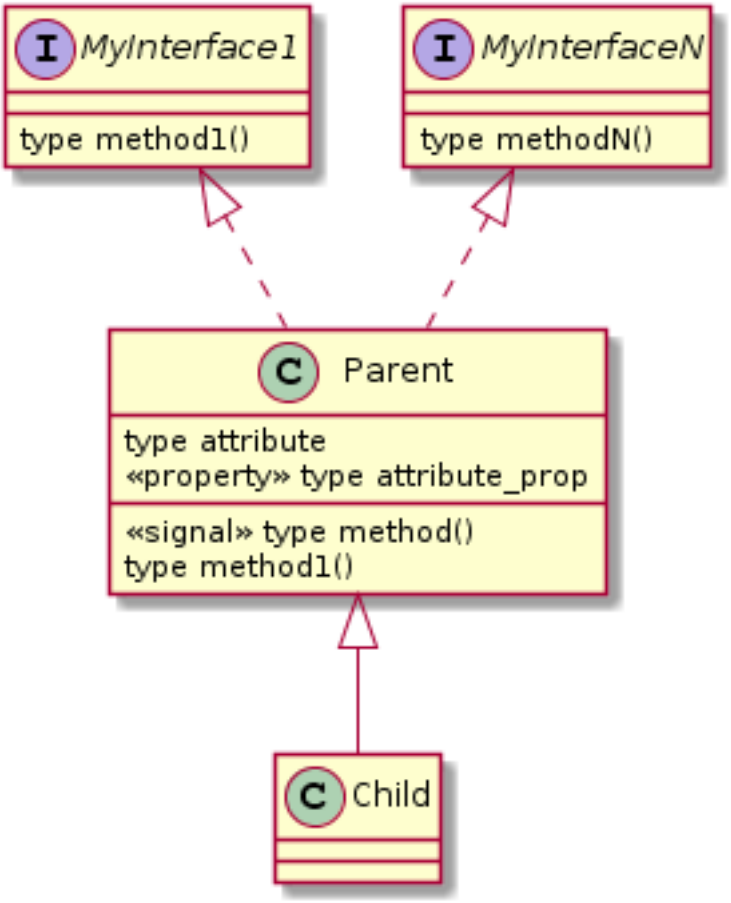
#	Tipo de dificultad	Nombre	Descripción
1	Propensión a errores	Inicialización de métodos polimórficos	Contrario a otros LPOO, a cada referencia de un método polimórfico debe asignársele su implementación en la función de inicialización de la clase.

#	Tipo de dificultad	Nombre	Descripción
2	Propensión a errores y obtención de información	Prototipado y asignación de métodos polimórficos redefinidos	Para crear el prototipo de un método ya definido por una clase padre y para poder asignarlo en la tabla virtual, se debe conocer exactamente cuál de los padres es el que definió el método por primera vez, además de que en caso de equivocarnos recibiremos una advertencia por parte del compilador, esto agrega una demora al programador que debe obtener esta información.
3	Repetición de información	LOCs por cada método polimórfico nuevo o redefinido	Si es nuevo se lo referencia al incluirlo en la tabla virtual y al definir su selector y tanto si es nuevo como redefinido: en su implementación, y al asignar la implementación a su referencia en la tabla virtual.
4	Repetición de información	Repetición en la información de la herencia	Se hacen varias referencias al padre para lograr una herencia
5	Dificultad de aprendizaje	Nuevas macros que aprender	Hay una importante cantidad de macros, funciones y especificaciones a aprender para utilizar correctamente el framework.

#	Tipo de dificultad	Nombre	Descripción
6	Repetición de código	repetición en cada clase	comparado con otros LPOO, hay mucho código repetido entre una clase y otra tan solo para definir la clase y su herencia, repitiendo varias veces el nombre de la clase como argumento de distintas macros.

Comparado incluso con otros frameworks puede apreciarse una gran cantidad de código de plantilla (en inglés boilerplate code) asociado al prototipado de una clase y sus métodos.

4.12.6 Propuesta de expresión en UML



Cualquier clase puede realizar una o varias interfaces y heredar de hasta una clase. Cualquier método en una clase puede definirse como polimórfico o no (mediante el atributo `isLeaf`), si es de clase o no (mediante el atributo `isStatic`), si es constante o no (mediante el atributo `isConst`), y si es abstracto o no (mediante el atributo `isAbstract`). Las interfaces pueden definir métodos como abstractos o no (o sea que ofrecen una implementación del método por defecto), como constantes o no y como de clase o no. Los atributos pueden ser de clase (mediante el atributo `isStatic`) y constantes (mediante el atributo `isConst`). Los atributos pueden poseer el estereotipo `property` con los siguientes valores etiquetados (correspondientes a los nombres de los parámetros pasados a la función `g_param_spec_int ()`): `nick`: apodo de la propiedad, `blurb`: descripción

de la propiedad, y un flag para todas las opciones de flags en el tipo enumerativo `GParamFlags`. El nombre y los valores minimos y maximos y por defecto de la propiedad son tomados de los definidos para el atributo. Un método puede poseer el estereotipo `signal` lo que lo hace un manejador de la misma (no analizaremos como modelar una señal sin manejador), los valores etiquetados del estereotipo serían (correspondientes a los nombres de los parámetros pasados a la función `g_signal_new()`): `name`: el nombre de la señal y un flag para todas las opciones de flags en el tipo enumerativo `GSignalFlags`. Hay más opciones para las señales y, por consiguiente, más análisis necesario de cómo modelarlas pero quedan fuera del alcance de esta tesis, ya que, como veremos, no presentan una dificultad respecto a la factibilidad de generar código desde tales modelos. Del mismo modo podría buscarse una propuesta de expresión para closures.

4.13 Dynace de Blake McBride

4.13.1 Introducción

Dynace tiene el objetivo de servir como un lenguaje de propósito general, con el poder de CLOS y SmallTalk, simple de usar como Objective-C, eficiente y lo más cercano posible a la sintaxis de C.
La filosofía de Dynace es focalizarse en la programación genérica que dista de la programación en lenguajes de tipado estático (como C++ o JAVA) (El manual de Dynace contiene una fuerte crítica al diseño que promueve trabajar con C++). Para eso se sostiene en un modelo de objetos dinámicos basado en CLOS y metaclases, lo que le provee una gran flexibilidad. Implementa funciones genéricas al estilo de CLOS, por lo que cualquier clase puede implementar estas funciones y la implementación a ser ejecutada es escogida en tiempo de ejecución de acuerdo al tipo del primer parámetro (Dynace sólo soporta funciones genéricas de rango 1, los multimétodos son funciones genéricas que se escogen de acuerdo al tipo de varios de sus parámetros en tiempo de ejecución).
Al contrario de los demás frameworks presentados en este estudio, Dynace

se define de acuerdo a la sintaxis entendible por su preprocesador, el cual no es el estándar de C. Con esto, aunque no utiliza la sintaxis de C, logra que sea muy amigable para el programador C, y el cuerpo de los métodos sí conserva la sintaxis de C. Las facilidades en tiempo de ejecución de Dynace lo hacen más costoso en el uso de memoria RAM y tiempo de ejecución (pero distan mucho de programar, por ejemplo, para una máquina virtual) por lo que no es recomendable para microcontroladores demasiado restringidos. Las clases en Dynace son codificados en un único archivo .d y el código generado sigue la norma ISO C89. Contiene una jerarquía de clases núcleo equivalente a la de SmallTalk, una biblioteca de clases completa, threads cooperativos así como soporte para threads nativos y gran cantidad de documentación para el principiante. Ha sido usado por más de 20 años en ambientes de producción.

4.13.2 Conceptos soportados

1. Encapsulamiento
2. Herencia (múltiple)
3. Polimorfismo
4. Duck typing
5. Manejo de excepciones
6. Metaclases
7. Recolector de basura
8. Programación clave-valor

4.13.3 Modelo de objetos

El análisis del modelo de objetos de Dynace excede el alcance de esta tesis.

4.13.4 Codificación

La especificación detallada de la sintaxis de Dynace se encuentra en su manual, aquí solo daremos una introducción suficiente para nuestro análisis.

Dynace define un único tipo para el uso de todas las clases, el tipo `object`. La implementación de una clase, con variable de instancia, de clase y especificación de una función de inicialización de clase (se usa en caso de que se necesiten instanciar recursos para la clase) se realiza con la palabra clave `defclass`:

```
//Parent.d
defclass  Parent{
    int instanceVariable; //Variable de instancia
    class:
    int classVariable;  //Variable de clase
    init: initParentClass; //función de inicialización de la
↪   clase
};
```

Las clases pueden tener visibilidad privada (por defecto) o pública, lo que permite referenciar a la estructura de instancia de la clase (de otra forma la misma está oculta).

En caso de herencia múltiple , el listado de clases sigue a la clase:

```
//MyClass.d
defclass MyClass : SomeClass, SomeOtherClass;
```

La implementación de genéricos puede especificar un solo método para varias funciones genéricas:

```
//MyClass.d
imeth int gMyGeneric, gGeneric2, gGeneric3 (char param)
{
return param;
}
```

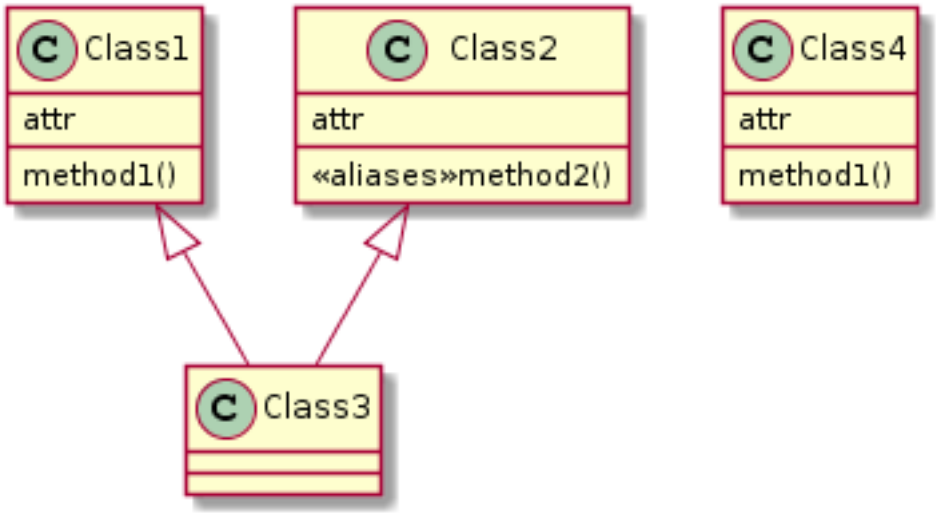
`imeth` indica que se trata de un método de instancia con verificación del tipo de los parámetros en tiempo de compilación. Las otras variantes son

`cmeth` para métodos de clase y sus variantes con lista de argumentos variable: `ivmeth` y `cvmeth`. El tipo de retorno en este caso es integer, pero en caso de omitirse por defecto es `object`, la clase padre de todas las clases. La simplicidad en la sintaxis muestra el poderío de Dynace como lenguaje de programación.

4.13.5 Dificultades en la codificación

Dynace no presenta ninguna de las dificultades de las que venimos estudiando salvo el hecho de tener que aprender su sintaxis para el prototipado de clases.

4.13.6 Propuesta de expresión en UML



La visibilidad de las clases y métodos sólo puede ser pública o privada. Los atributos no tienen atributo de visibilidad. Todos los métodos públicos son polimórficos, por lo que el atributo `isLeaf` es ignorado. Puede indicarse si el método es de clase o no (mediante el atributo `isStatic`), si es constante o no (mediante el atributo `isConst`). Existe una forma de exigir que un método sea implementado mediante el método `gSubclassResponsibility()` que arroja una excepción si un método no está siendo redefinido por un objeto hijo, con esto pueden definirse métodos abstractos mediante el atributo `isAbstract`. Si la implementación del método es el mismo para varios

otros métodos, entonces el estereotipo `aliases` es agregado con el valor etiquetado `names` donde se indican los nombres de dichos métodos separados por coma. Los atributos pueden ser de clase (mediante el atributo `isStatic`) y constantes (mediante el atributo `isConst`). Cabe aclarar que por más que las clases no tengan una ascendencia en común pueden implementar los mismos métodos que son llamados con los mismos mensajes (en el ejemplo tanto una instancia de `Class1` como de `Class4` pueden responder al mensaje `method1()`).

4.14 COS de Laurent Deniau

4.14.1 Introducción

COS utiliza el preprocesador ISO C99 para generar código ISO C89. El diseño y la sintaxis de COS están fuertemente inspirados en Objective-C y CLOS (Common Lisp Object System), según el autor, uno de los modelos de objetos más flexibles jamás desarrollados, y en menor medida por Cecil, Dylan, Haskell, Python, Slate y SmallTalk . COS impone la encapsulación fuerte y la separación de incumbencias (en inglés concern separation) a través de su modelo de objeto abierto, que permite usar y extender los componentes de COS (por ejemplo, clases) definidos en bibliotecas compartidas sin tener el código fuente. Este modelo de objetos abierto exige una etapa extra de linkeo para recolectar símbolos distribuidos en el código, esto se realiza a través de los makefiles que acompañan al framework. El kernel de COS es de sólo 7000 líneas de código fuente y cumple muy bien los cinco principios a los que apunta: simplicidad, flexibilidad, extensibilidad, eficiencia y portabilidad. Posee un modelo de objeto uniforme, y permite la programación con contratos y closures.
El diseño de COS está optimizado para proporcionar una implementación portátil y eficiente, especialmente en sus dos características principales: los multimétodos, así como el reenvío de mensajes genéricos (es decir, la delegación). El envío de mensajes en COS es de 1,7 a 2,3 veces más lento que la llamada a una función indirecta (o sea a través de punteros) y

aproximadamente de 1,2 a 1,5 veces más rápido que el envío de mensajes en Objective-C. El reenvío de mensajes en COS es tan rápido como el envío de mensajes y aproximadamente de 40 a 80 veces más rápido que el reenvío de mensajes en Objective-C, que, además, tiene fuertes limitaciones en los valores devueltos. Estos dos conceptos junto a su modelo de objetos permiten implementar fácilmente: mensajes de orden superior (high order messages), predicado de clase (class-predicate dispatch), herencia múltiple, herencia dinámica, clases dinámicas, modelo de objeto adaptativo, reflexión y gestión avanzada de memoria. COS logra los principios de simplicidad, flexibilidad y extensibilidad, así como los lenguajes de scripting convencionales existentes (por ejemplo, PHP, Python, Ruby, Lua) al tiempo que mantiene la eficiencia y la portabilidad en el rango de C(Deniau 2009). No es recomendado para ser utilizado en sistemas altamente restringidos con muy poca memoria RAM ya que la caché para optimizar el reenvío de mensajes puede ocupar algunos megabytes (esto puede ser configurado dentro de cierto rango).

4.14.2 Conceptos soportados

1. Encapsulamiento
2. Herencia
3. Polimorfismo
4. Multimétodos
5. Reenvío de mensajes (message forwarding) / Delegados
6. Manejo de excepciones
7. Metaclases
8. Programación clave-valor

4.14.3 Modelo de objetos

Un análisis detallado del modelo de objetos de COS excede el alcance de esta tesis. COS sólo soporta herencia simple basado en la herencia de estructuras de instancia, `Object` es la clase base de todas las clases. La codificación bajo COS está basada en la definición de tres componentes básicos: las

clases o tipos, las funciones genéricas y los métodos, que definen los tipos de argumento para una función, puede existir varios métodos para una sola función. Un método es una implementación o especialización de una función. Las funciones son llamadas en tiempo de ejecución y la implementación a ser ejecutada es decidida en base a los tipos en tiempo de ejecución de sus argumentos (en COS hasta cinco argumentos pueden ser tomados en cuenta para esta decisión), las implementaciones cuyos argumentos contienen la misma especialización que los de la llamada son ejecutados, sino se escogen implementaciones con argumentos menos especializados (o sea que sean clases padre respecto a los argumentos de la llamada) priorizando los tipos de los primeros argumentos,[1] en caso de no encontrarse una implementación apropiada la función genérica `gunrecognizedMessageN ()` en llamada, donde N es la cantidad de argumentos que son tomados en cuenta y por defecto lanza una excepción. Esta última función da lugar a reenviar la llamada a alguna otra clase (message forwarding). Esto nos lleva a un modelo que en vez de tablas virtuales para cada clase tenemos tablas de métodos para cada función.

4.14.4 Codificación

El modelo de objetos abierto de COS permite definir cada uno de los elementos que ejemplificaremos en un archivo distinto, es decir, cada definición de clase, método o propiedad puede estar en un archivo distinto. Esto permite una separación de incumbencias muy bueno para el buen diseño de un sistema.

4.14.4.1 Funciones genéricas

Se declaran en una interfaz (.h) de la siguiente manera:

```
//gfunction.h
#include <cos/Object.h>
```

[1] Para una descripción más detallada leer **Methods specialization** en la documentación de COS.

```
defgeneric(void, gfunction, self, arg1);
```

Luego de la macro `defgeneric` el primer argumento es el tipo a retornar por la función, luego el nombre de la función genérica (por convención comienza con g) y luego los argumentos de la función. Los argumentos `self` y `arg1` son llamados tipos abiertos, ya que cada tipo puede variar de acuerdo a la implementación. Puede definirse funciones genéricas que también posean tipos cerrados por ejemplo:

```
//gfunction.h
#include <cos/Object.h>

defgeneric(void, gfunction3, _1 ,(int)i);
```

4.14.4.2 Clases

Las clases poseen visibilidad de declaración y de implementación como ya hemos visto en otros frameworks, pero a diferencia de ellos no es necesario escribir otro archivo de interfaz adicional, el único que se debe escribir es el que corresponde a la visibilidad de implementación.

4.14.4.2.1 interfaz (.h) Sólo es necesario definir los atributos de la clase, su nombre y su clase padre.

```
//Child.h
#include "Parent.h"

defclass(Child, Parent)
    int attribute;
    OBJ class1;
endclass
```

En caso de que una clase descienda de `Object` el segundo argumento de `defclass()` es omitido.

4.14.4.2.2 implementación (.c) Mostraremos la instanciación de la clase junto a la implementación de algunos métodos.

```
//Child.c
#include "Child.h"
#include "gfunction.h"
#include "gfunction2.h"

#include <cos/gen/message.h>
```

La siguiente declaración instancia la clase:

```
makclass(Child, Parent);
```

Antes de poder utilizar una clase externa, la misma debe ser declarada con `useclass()`:

```
useclass(Class1);
```

Los siguientes son los métodos de inicialización y destrucción de la clase:

```
defmethod(OBJ, ginit, Child)
  self->class1 = gnew(Class1);
  retmethod(_1);
endmethod

defmethod(OBJ, gdeinit, Child)
  grelease(self->class1);
  retmethod(_1);
endmethod
```

El siguiente método es una especialización de la función genérica `gfunction()`, a partir del tercer argumento de `defmethod()` se indican los tipos específicos:

```
defmethod(void, gfunction, Child, Object)
  /*...*/
endmethod
```

Es posible asignar una misma implementación a varias funciones genéricas, esto se hace mediante los alias, a continuación se asigna la implementación anterior de `gfunction()` a `gfunction2()`:

```
defalias(void, (gfunction )gfunction2, Child, Object);
```

Se puede definir un método rodeador (en inglés around) que es ejecutado en vez del método que rodea y puede definir acciones antes y después del mismo:

```
defmethod(void, (gfunction), Child, Object)
    /*antes de gfunction*/
    next_method(self1,self2); //llamada a gfunction()
    /*después de gfunction*/
endmethod
```

El siguiente es un ejemplo de reenvío de mensajes a `class1`:

```
defmethod(void, gunrecognizedMessage1, Child)
        if(gunderstandMessage1(self->class1,_sel) == True)
            forward_message(self->class1); // delegación
endmethod
```

4.14.4.3 Propiedades

Las propiedades en COS son parte de su biblioteca base, es decir, no tienen un soporte especial desde su modelo de objetos sino que son soportadas tan solo con los multimétodos.

4.14.4.3.1 Definición

```
//value.h
#include <cos/Property.h>

defproperty( value );
```

4.14.4.3.2 Definición como propiedad de clase

```
//ChildProperties.c
#include "Child.h"
#include "value.h"
#include <cos/Number.h>
#include <cos/gen/value.h>


static OBJ int2OBJ(int val) {
return  aInt(val);
}


defproperty(Child, (attribute)value,int2OBJ,gint);
```

En la definición anterior de la propiedad el atributo `attribute` es accedido a través de la propiedad `value`.

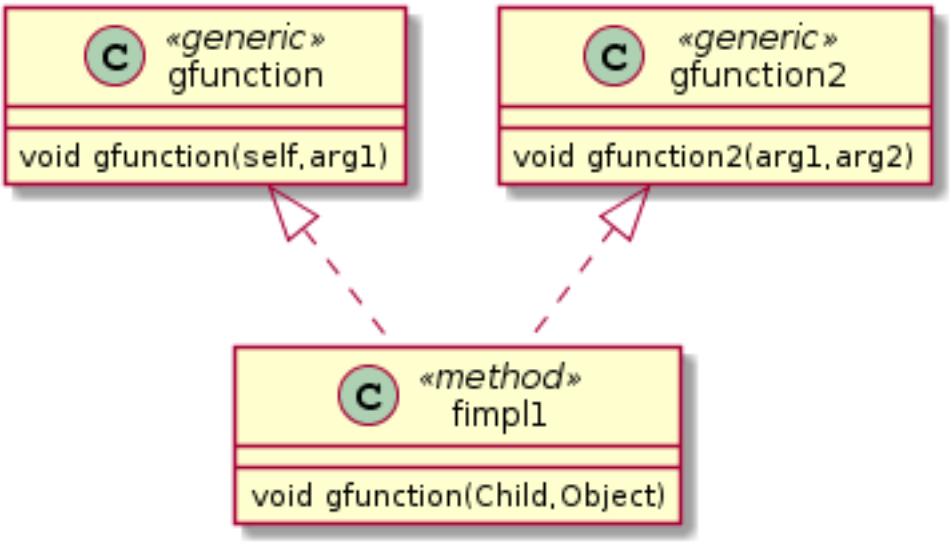
4.14.5 Dificultades en la codificación

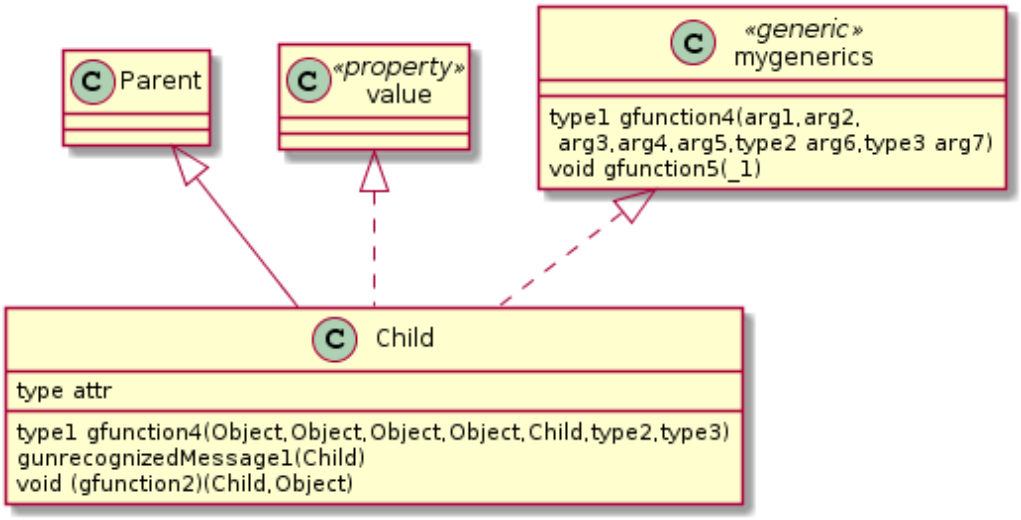
Muy pocas dificultades podemos mencionar para el prototipado bajo este framework. Primero el nuevo lenguaje debe aprenderse. Segundo podemos notar la duplicación de información en la declaración de una clase y su herencia en las macros `makclass()` y `defclass()`. Una dificultad más encontrada que no entra dentro de nuestro estudio pero vale la pena mencionar para trabajos futuros es que una biblioteca de macros tan

compleja produce que se arrojen una gran cantidad de mensajes de error al introducir un error de sintaxis al codificar bajo la especificación del framework, esto dificulta encontrar dicho error, por lo que una herramienta previa que analice errores en el uso del framework podría ser de gran utilidad.

4.14.6 Propuesta de expresión en UML

Cómo modelar los multimétodos en UML requiere un análisis profundo ya que fueron concebidos luego de que UML haya sido creado y puede no ser un buen lenguaje de modelado para los mismos. Esta propuesta puede ser tomada como puntapié para el modelado bajo otros lenguajes que soportan multimétodos como CLOS.





La base de este modelado es relacionar clases (sujetos) y funciones genéricas (verbos) a través de métodos (especialización).

Las clases poseen atributos y métodos (que explicaremos mejor a continuación). Pueden heredar de una única clase y en caso de no indicarse una herencia heredan de la clase `Object`. Las funciones genéricas se definen como operaciones dentro de una clase con el estereotipo `<<generic>>` (esto permitiría que una o varias funciones genéricas sean definidas en un mismo archivo). Los argumentos de las funciones genéricas que no poseen un tipo son tipos abiertos y se admiten hasta cinco (que deben además ser los primeros), los argumentos que posean tipo serán tipos cerrados. Los métodos pueden ser definidos tanto dentro de una clase como fuera de ellas dentro de una clase con el estereotipo `<<method>>` (esto permite que sean definidas en un archivo aparte del de alguna clase). Los métodos pueden realizar varias funciones genéricas (en el ejemplo el método `gfunction4()` en `Child` realiza `gfunction4()`), en caso de realizar más de una función genérica con la misma signatura, las mismas son alias entre sí, si el método tiene el mismo nombre que alguna de ellas, la misma es definida (`defmethod`) y el resto son considerados alias (`defalias`). Los métodos deben contener la misma cantidad de argumentos que la función genérica que realizan y deben definir todos sus tipos. En el caso de que un método no realice una función genérica se considera que realiza a la función genérica que posee su nombre (en el ejemplo sería el caso de `gunrecognizedMethod1()`), de ser

necesario, una dependencia de tipo usage deberá hacer visible la función genérica para el método (relacionando a ellos dos o a sus contenedores). Un método around no agrega ninguna regla adicional a las anteriores para poder modelarlo, tan sólo se precisa un método con el nombre de la función genérica entre paréntesis (En el ejemplo es el caso de `(gfunction2)()` dentro de `Child`).
Las propiedades se modelan como clases (ya que internamente es lo que son) con el estereotipo `<<property>>` y pueden relacionarse con un atributo que las realice (en el ejemplo el atributo `attr` de `Child` está realizando a la propiedad `value`).

4.15 Conclusiones

Hemos visto una gran cantidad de frameworks muy dispares entre sí en sus modelos de objetos, algunos tan solo dando soporte al polimorfismo y otros con modelos comparables con los de C++ (OOPC), JAVA (OOC-S y OOC de Miseta), Small Talk (Dynace) y otros con CLOS (Dynace y aún más COS). Sin embargo, respecto a lo que atañe a la dificultad en el prototipado bajo cada uno de ellos podemos sacar varias conclusiones:
Los frameworks que se valen de herramientas externas en el proceso de convertir código fuente en artefactos como bibliotecas o ejecutables pueden sortear las dificultades del prototipado de objetos en C, en general la herramienta más utilizada es un preprocesador como en el caso de Dynace (también OOC de Schreiner y para el caso de GObject existen compiladores fuente a fuente como Vala y Genie). La desventaja principal que se introduce al utilizar este enfoque es que el editor que se utilice no entenderá el código que estamos escribiendo por lo que herramientas de búsqueda, autocompletar o refactoring no nos servirán. Además, un nuevo lenguaje -y no estándar- debe aprenderse. Por otro lado, como se vió en el caso de Dynace, un preprocesador externo puede dejarnos tan sólo con la dificultad un lenguaje muy reducido en nuevas palabras respecto del lenguaje C. Entre los frameworks que se valen de herramientas externas COS se distingue de los demás valiéndose tan sólo de una etapa extra de linkeo y una increíble biblioteca de macros C99, esto le permite utilizar C puro sin adición de

dificultades significativas en el prototipado. Creemos que este framework debe tomarse como objeto de investigación para futuros frameworks que deseen implementar un modelo de objetos distinto y quizás uno aplicable a los sistemas embebidos. Respecto del resto de los frameworks que no se valen de herramientas externas (de los estudiados: Ben Klemens, SOOPC, OOC de Tibor Miseta, OOPC, OOC-S y GObject), a pesar de que el procesador de C y bibliotecas puede ayudar, precisan una gran cantidad de código repetitivo y repetición de información para el prototipado de clases. La forma en que el código crece en líneas al adicionar métodos a las clases es de aproximadamente 3 a 5 veces más respecto de utilizar un LPOO. Justamente algunos de estos frameworks son los más apropiados para sistemas altamente restringidos. Pero como ya hemos dicho, evaluar el framework en soledad no es algo correcto. Todavía podemos introducir herramientas dentro del tool suite del programador que disminuyan dichas dificultades y nos permitan seguir trabajando bajo el lenguaje C. Nuestra propuesta es un generador de código desde diagramas UML a introducir en el siguiente capítulo.

Capítulo 5

Estado del arte en generadores de código C desde diagramas UML

5.1 El uso de modelos para la generación de código C orientado a objetos

Una de las herramientas adecuadas para conseguir un toolsuite que simplifique la codificación OO en C es un generador de código desde UML (Hendrickx 2004), sobre todo en lo que respecta al prototipado de clases que fue el centro de atención en las dificultades que exploramos en el capítulo anterior . A través de la generación automática de código se reducen errores y esfuerzo (M. Maranzana & Bernier 2004), esto es especialmente valedero para cuando hablamos en COO. Todavía hay mucho por desarrollar en esta disciplina de la generación automática de software que se encuentra "inmadura por sus restricciones o pobre en su optimización" (Murillo 2009). Las aplicaciones para generar diagramas UML por sí solas pueden ser útiles pero su verdadera utilidad se da cuando tienen un generador de código automático asociados a ellos y, al mismo tiempo, son costeables por el presupuesto del desarrollo (Samek 2012). En esta sección mostraremos que esto es posible con herramientas de código abierto.

5.2 Estado del arte en generadores de código C desde diagramas de clase UML

Si bien existen herramientas para generar código C desde el lenguaje UML, todavía no existe un generador de código C que se ajuste correctamente al paradigma de la orientación a objetos y tampoco para ninguno de los frameworks estudiados.
A continuación introduciremos la capacidad de generación de código orientado a objetos por las siguientes herramientas populares: Enterprise Architect (Architect 2017), Rational Rhapsody (IBM 2010), Astah (Anón 2019) y UML Generators (Eclipse 2016) (esta última no fue seleccionada por su popularidad sino más bien porque será la base del generador de código que propondremos).

5.2.1 Enterprise Architect

La siguiente cita fue extraída de la página de Enterprise Architect respecto de cómo modelar COO en Enterprise Architect:

> Generación de código C orientada a objetos para el modelo UML
>
> La idea básica de implementar una Clase UML en código C es agrupar la variable de datos (atributos UML) en un tipo de estructura; esta estructura se define en un archivo .h para que pueda ser compartida por otras Clases y por el cliente que se refirió a ella.
>
> Una operación en una clase UML se implementa en código C como una función; el nombre de la función debe ser un nombre completamente calificado que consiste en el nombre de la operación, así como el nombre de la Clase para indicar que la operación es para esa Clase. [...]
>
> La función en el código C también debe tener un parámetro de referencia para el objeto Clase: puede modificar las opciones 'Referencia como parámetro de operación', 'Estilo del parámetro

de referencia' y 'Nombre del parámetro de referencia' en la página 'Especificaciones de C' para admitir esto parámetro de referencia.

Limitaciones de la Programación Orientada a Objetos en C

- Sin asignación de visibilidad para un atributo: un atributo en una Clase UML se asigna a una variable de estructura en el código C, y su alcance (privado, protegido o público) se ignora.
- Actualmente se ignora una Clase interna: si una Clase UML es la Clase interna de otra Clase UML, se ignora al generar el código C.
- Se ignora el valor inicial: el valor inicial de un atributo en una Clase UML se ignora en el código C generado.
  (Architect 2017)

Como podemos ver, sólo una noción básica de encapsulamiento es soportada por este generador de código, ninguna especificación de herencia o polimorfismo es dada, quizás lo mejor que podríamos hacer es incluir una estructura padre dentro de otra hija para obtener herencia, y declarar punteros a función como métodos polimórficos, pero estaríamos abandonando ya la sintaxis estándar de UML para dichos conceptos.

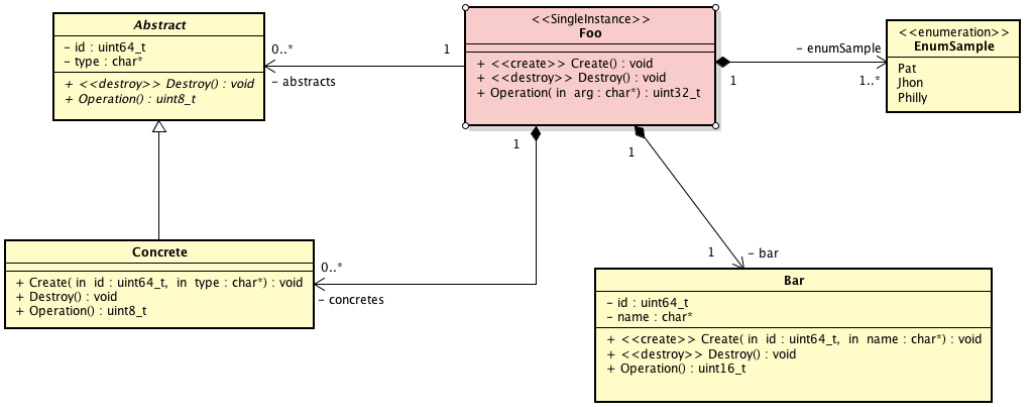
5.2.2 Rational Rhapsody

Douglass (Douglass 2010) propone representar las clases en estereotipos UML de tipo `<<File>>` que pueden procesarse en Rhapsody para obtener archivos .c y .h colocando las variables, declaraciones, funciones y prototipos en el archivo correspondiente de acuerdo a la visibilidad definida en el estereotipo. De nuevo tenemos los mismos inconvenientes que para con la herramienta anterior.

5.2.3 Astah

Astah posee un plugin llamado UML2c concebido con el propósito de facilitar la codificación bajo la especificación de Grenning (Grenning 2011) para obtener código orientado a pruebas. El plugin además de generar código C, genera el código de mocks para ser utilizados con Google Test (Google 2019).

Esta es la primer herramienta que facilita en alguna medida el uso del polimorfismo bajo lenguaje C. A continuación se presenta un diagrama de muestra del plugin con los distintos elementos de los cuales genera código.



Como puede apreciarse existe una relación de herencia entre la clase padre `Abstract` y la clase hija `Concrete`, `Abstract` está definida como una clase abstracta en el diagrama.

El generador de código genera 3 archivos por cada clase, uno de implementación (.c), y 2 de interfaz (.h). En uno de los archivos de interfaz, si la clase es abstracta, entonces se crea una estructura con punteros a función que corresponden a las operaciones definidas para esa clase, esta estructura serviría para instanciar una tabla virtual para la clase y sus derivadas. Luego es definida una estructura de instancia de clase cuyo primer miembro es un puntero a la tabla virtual y a continuación los atributos de la clase.

Las clases hijas no extendien a la tabla virtual por más que también sean abstractas, por lo que esta facilidad solo sirve para 1 nivel de herencia.

Los demás archivos de la clase hija son esqueletos de las operaciones definidas, por lo que no existen más facilidades que esta.

La clase hija no contiene ninguna referencia a la clase padre por lo que tampoco brinda facilidades en la implementación del polimorfismo (instanciación de la tabla virtual, asignación de su referencia en una instancia, asignación de implementaciones de funciones).

A continuación presentamos el código generado para la clase `Abstract` y `Concrete`

```
//Abstract.c
#include "Abstract.h"

void Abstract_Destroy(Abstract self)
{
    /* TODO */
}

uint8_t Abstract_Operation(Abstract self)
{
    /* TODO */
}

//Abstract.h
typedef struct AbstractStruct *Abstract;


void Abstract_Destroy(Abstract self);
uint8_t Abstract_Operation(Abstract self);

#include "AbstractPrivate.h"

//AbstractPrivate.h
typedef struct AbstractInterfaceStruct *AbstractInterface;


struct AbstractStruct {
```

```
    AbstractInterface vtable;
    uint64_t id;
    char *type;
};


typedef struct AbstractInterfaceStruct {
    void (*Destroy)(Abstract self);
    uint8_t (*Operation)(Abstract self);
};

//Concrete.c
#include "Concrete.h"

void Concrete_Create(Concrete super, uint64_t id, const char
↪   *type)
{
    /* TODO */
}

void Concrete_Destroy(Concrete super)
{
    /* TODO */
}

uint8_t Concrete_Operation(Concrete super)
{
    /* TODO */
}

//Concrete.h
typedef struct ConcreteStruct *Concrete;
```

```
void Concrete_Create(Concrete super, uint64_t id, const char
↪  *type);
void Concrete_Destroy(Concrete super);
uint8_t Concrete_Operation(Concrete super);

#include "ConcretePrivate.h"

//ConcretePrivate.h
struct ConcreteStruct {
};
```

Como puede apreciarse, la ayuda que provee este generador de código es muy reducida dejando gran cantidad de trabajo al programador tan solo para el prototipado.

5.2.4 UML Generators

UML generators es un proyecto de código abierto que se compone de varios generadores de código desde UML (algunos implementan la ingeniería inversa). Los generadores están escritos en Acceleo (Acceleo 2019). Acceleo implementa el estándar de transformación de modelo a texto del MOF (model object facility) perteneciente al OMG (object model group). El mismo utiliza OCL (object constraint language) para obtener información del modelo a convertir en texto.
UML Generators posee 2 generadores de código desde UML a lenguaje C. Uno de ellos (C generator) posee una aplicación de ingeniería inversa para generar los modelos desde el código y el otro (UML to Embedded C generator) está orientado a su uso en sistemas embebidos.

5.2.4.1 C generator

C generator utiliza la notación de UML pero define un lenguaje propio para modelar el código que el programador C está acostumbrado a escribir. Así

por ejemplo, una clase privada representa un archivo de implementación (.c), una interfaz un archivo de interfaz (.h) y una clase pública a ambos. Esto ya nos indica que ni siquiera el concepto de clase de la OO es facilitado por este modelado.

5.2.4.2 UML to Embedded C generator

UML to Embedded C generator es una contribución de Spacebel[1] a UML Generators. El objetivo de este generador ha sido:

1. Generación de código repetible y confiable.
2. Preservación de los campos de implementación.
3. Diseño detallado altamente documentado.
4. Cumplimiento de las directrices comunes del lenguaje C en el dominio aeroespacial.
5. Trazabilidad de la especificación (requisitos) en las fuentes.

Para extender la semántica de UML posee un perfil (en inglés profile) UML que posee los estereotipos que precisa para su modelado.
El modelado y la generación de código orientado a objetos soportados son muy similares a los vistos en Enterprise Architect y Rhapsody. Una clase representa un TDA (tipo de dato abstracto) cuyos atributos son declarados en una estructura de instancia de la clase, las operaciones públicas definidas en la clase pasan a ser funciones declaradas en el archivo de interfaz donde se declara el TDA y las privadas son definidas en el archivo de implementación. La relación de generalización no tiene significado para este generador por lo que los conceptos de herencia no son facilitados.

5.3 Resumen

La conveniencia de expresar modelos aparte del código ha impulsado el desarrollo de generadores de código C desde diagramas de clase UML para

[1] https://www.spacebel.be

expresar la estructura contenida en dicho código. Debido a que C no es un LPOO mientras que UML es un lenguaje inherentemente orientado a objeto ha llevado a un recorte del lenguaje o redefinición de su semántica para ser utilizado junto con C. En general, de los conceptos de la OO, el encapsulamiento es el único concepto soportado por estos generadores. El único generador que incorpora en alguna medida herencia y polimorfismo es el uml2c de astah, la herencia soportada es de hasta un nivel (un padre y varios hijos que no pueden a su vez ser padres de otras clases) y el código generado deja mucho de la implementación al programador (inclusión de la clase padre en la del hijo, inicialización de la tabla virtual, inicialización de un objeto con su clase padre).

El estado actual de los generadores de código C desde UML nos muestra que un generador que implemente varios conceptos de la orientación a objetos y que se encargue en su totalidad del prototipado de los artefactos relacionados con la orientación a objetos (clases, interfaces, etc.) está vacante.

Capítulo 6

Implementación de generadores de código C embebido, eficiente y orientado a objetos desde diagramas de clase UML

6.1 Punto de partida

En base a nuestro estudio en el capítulo anterior hemos decidido investigar la factibilidad de generar código desde UML para los frameworks estudiados (de acuerdo a sus propuestas de expresión presentadas) con el lenguaje Acceleo debido a que es el lenguaje basado en un estándar del OMG (Otros lenguajes utilizados para transformar modelos en texto son Xpand(XPand 2019) y Xtend(Xtend 2016)), además, este estudio permitirá extender el generador de código UML to Embedded C generator de UML generators que fue realizado para el dominio de aplicaciones que han sido el objetivo de varios de los frameworks estudiados, sin por eso ser inadecuado para su uso en otros dominios.
Acceleo permite delinear en el código generado los lugares donde el programador puede codificar sin que el código sea borrado por el generador de código.

6.2 Factibilidad

Los lenguajes de modelo a texto, y en particular Acceleo, ya tienen su experiencia en haber sido usados para implementar generadores para distintos LPOO, para eso fueron concebidos.
La pregunta está en si estos lenguajes utilizados junto con el metamodelo de UML nos permitirán generar código para los frameworks de COO estudiados.
Esta pregunta puede no tener una respuesta trivial ya que el código escrito bajo estos frameworks requiere de información adicional, obtenible del modelo mediante consultas personalizadas, del que se requiere en los LPOO, por ejemplo qué clase fue la primera en declarar cierto método para determinar el tipo del argumento `self` (si es el caso para ese framework), o poder a partir de esa información codificar la inicialización en ROM de una tabla virtual con los métodos ordenados en forma correcta en dicha tabla bajo el estándar ISO C89 (a partir del ISO C99 la inicialización de una estructura se puede codificar con el nombre de sus miembros y no con el orden de los mismos). Los únicos frameworks que no presentan dificultades de este tipo son Dynace y COS.

6.3 Verificaciones

Para verificar que las consultas sean correctas se ha analizado el código generado a partir de ellas, desde un modelo UML. En la mayoría de los casos el generador utilizado ha sido el desarrollado a partir del *UML to Embedded C generator* de *UML Generators* para los frameworks SOOPC, OOC de Miseta y Dynace.
Para cada una de las siguientes secciones de generación de código se encuentran tanto el modelo como el código utilizado.

6.4 Mejoras en el código para facilitar la generación de código

Si bien nos limitaremos a generar código para los seis frameworks que se han vuelto el objetivo de esta tesis (Ben Klemens, SOOPC, OOC-S, OOPC , OOC de Miseta y OOC de Schreiner) no nos contendremos en proponer mejoras en las especificaciones de codificación de dichos frameworks que faciliten el mapeo entre modelos UML y dicho código. Esto será útil ya que simplificará a los generadores de la nueva especificación e incluso facilitará en alguna manera al programador de clases. Estas mejoras pueden tener su contrapartida, quizás la adición de más líneas de código o el cambio de estándar de codificación C (por ejemplo de ISO C89 a C99).

6.5 Generación de código

Hay algunas cuestiones que tomaremos a priori (acertadamente) como posibles a resolver sin problemas bajo Acceleo y que son comunes a otros lenguajes de programación orientado a objetos: obtener toda la información de una clase, sus atributos y métodos y sus relaciones inmediatas con otros. Teniendo esto en cuenta sólo deberemos investigar la factibilidad y facilidad de obtener información no inmediata a la clase o artefacto a generar. Las siguientes secciones tratarán esos puntos.

6.5.1 Plantilla para generar la tabla virtual en SOOPC

Lo primero que intentaremos resolver es generar el código de la instanciación de la tabla virtual de una clase escrita para SOOPC. Para esto deberemos poder llegar hasta el final de la cadena de herencias hasta la primer clase que declare método polimórficos, cada clase hija generará llaves nuevas debido al anidamiento de estructuras además de agregar métodos nuevos.

Lo primera información que buscaremos obtener es toda la jerarquía de clases

(por tratarse de herencia simple esto puede obtenerse en una lista). Esto podemos hacerlo utilizando una consulta (query) recursiva.

```
[query
public
getEntireClassesHierarchechy(aClass: Class) : OrderedSet (
↪  Class ) =
if aClass<>null then
    aClass.superClass.getEntireClassesHierarchechy() ->
↪  asOrderedSet()
        ->prepend(aClass)
else
    OrderedSet{}
endif
 /]
```

Luego de que podemos obtener esta lista podemos fácilmente obtener la lista de clases padre de una clase excluyendo a la clase misma de la lista. Esta consulta la llamamos `getAncestors()`.
Luego para cada clase deberemos saber si los métodos virtuales que definen se tratan de una redefinición polimórfica de los mismos (por lo que está declarada en una tabla virtual anterior) o es la primera definición de la misma por lo que genera una referencia nueva en su tabla virtual.
La siguiente consulta nos resuelve eso:

```
[query
public
isAMethodRedefinition(o: Operation) : Boolean =
o.class.getAncestors().allNotFinalOperations()
    ->exists(o1:Operation |redefine(o1,o))/]
```

En Acceleo al aplicar una función sobre un contenedor (como el `OrderedSet` devuelto por `getAncestors()`) con el símbolo `->` entonces el contenedor mismo es computado devolviendo cualquier otro tipo, al aplicar una función

con . entonces la función es operada sobre cada miembro del contenedor devolviendo un contenedor nuevo (conteniendo los tipos devueltos por la función).
De esta manera la consulta anterior se lee de la siguiente manera: para la operación `o` se obtiene la clase la contiene, de esa clase se obtiene la lista de sus ancestros, luego se obtiene un contenedor reemplazando cada ancestro por sus métodos polimórficos (devueltos por la función `allNotFinalOperations()`) luego se interroga si dentro del conjunto de los métodos polimórficos de las clases padre existe (`exists()`) un método que tiene el mismo nombre que el método `o` (`redefine(o1,o)`).
De esta manera podemos escribir fácilmente otra consulta que nos ayude a obtener los métodos polimórficos definidos por primera vez por la clase que llamaremos `getNotRedefine ()`.
La última información que deberemos poder obtener es cual es la última implementación disponible de cierto método para asignarla a la tabla virtual. Para eso usamos la recursividad comenzando desde las operaciones de la clase yendo hacia la de su ancestro.

```
[template
public
getLastClassDefinerName(aClass: Class ,aMethod: Operation)]
    [if (aClass.ownedOperation->exists(m:Operation |
        m.redefine( aMethod ) ))]
        [aClass.name/]
    [else]
    [if aClass.hasSuperclass()]
        [aClass.getSingleInheritanceSuperClass().
            getLastClassDefinerName(aMethod)/]
    [/if]
    [/if]
[/template]
```

`getLastClassDefinerName()` recibe la clase de la que se quiere saber cual es la última clase en su jerarquía que implementa el método pasado.

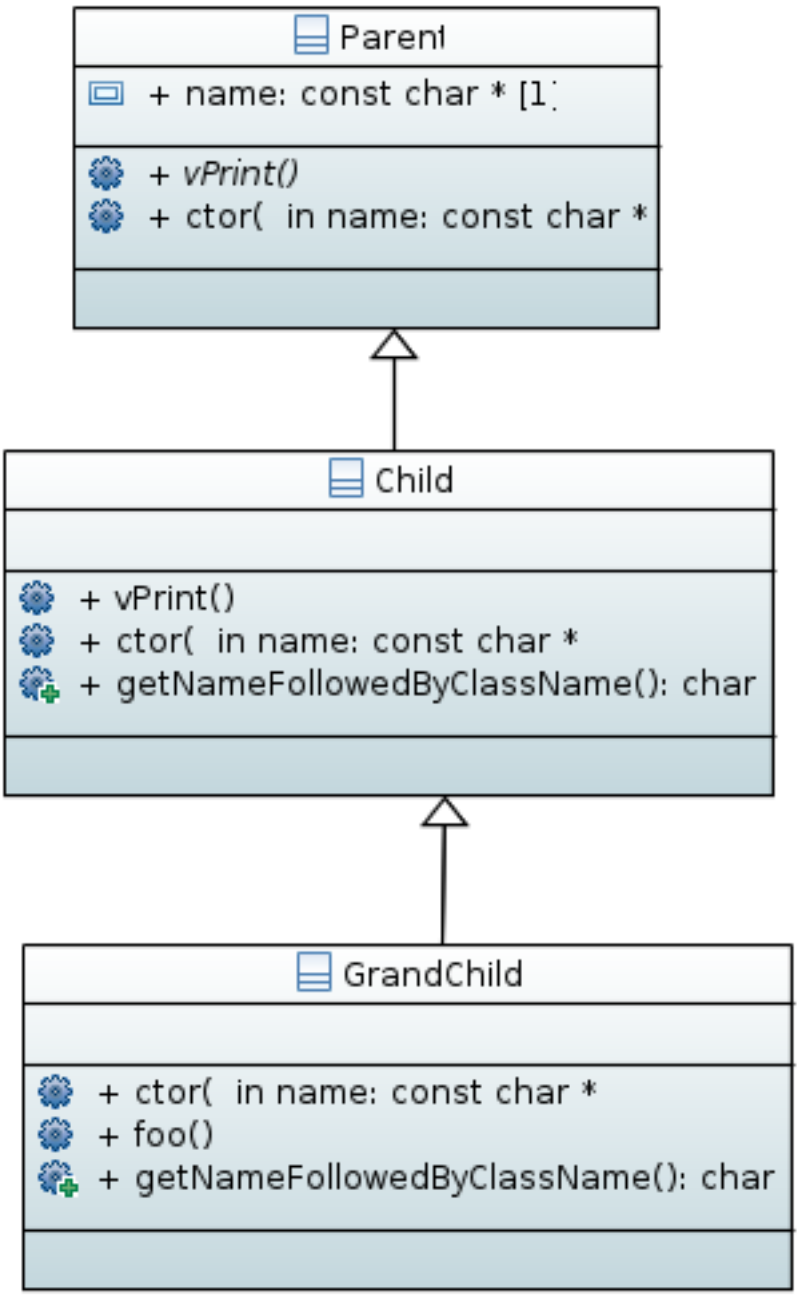
Luego que tenemos todas estas consultas podemos crear una plantilla para instanciar la tabla virtual de una clase:

```
static [aClass.name/]Vtbl const vtbl={
[for (c : OclAny | aClass.getAncestors()) ]{[/for]
[for (c : Class |
↪  entireClassesHierarchechy(aClass)->reverse()) ]
[let seqOp2 : Sequence(uml::Operation) =
    getNotRedefine ( getOwnedOperations ( c ) ) ]
[for (o : Operation |  seqOp2 ) separator (',')]
    [aClass.getLastClassDefinerName(o)/]_[o.name/]_
[/for]
[/let]
[if (c <> aClass)]      },[/if]
[/for]
};
[/if]
```

Generar el código de declaración de la clase virtual es mucho más simple ya que solo debe conocerse la clase padre y los nuevos métodos polimórficos (`getNotRedefine ()`).

6.5.1.1 Verificación

Utilizando el siguiente modelo UML:



Donde `vPrint()` es definida como un método abstracto y `getNameFollowedByClassName()` como un método polimórfico (`isLeaf` falso), la plantilla anterior genera la

siguiente tabla virtual para la clase `GrandChild`:

```
static GrandChildVtbl const vtbl={
{{
    Child_vPrint_           },
    GrandChild_getNameFollowedByClassName_          },

};
```

Puede notarse el uso correcto de llaves para asignar la función en el nivel de anidamiento de estructura correspondiente (`vPrint()` en la estructura `ParentVtbl` y `getNameFollowedByClassName()` en `ChildVtbl` que incluye a la anterior y a la vez es contenida por `GrandChildVtbl`) Por cuanto que la última clase en implementar `vPrint()` fue `Child` (obtenido mediante la consulta `getLastClassDefinerName()`) la misma es la utilizada.

6.5.1.2 Mejoras

Como puede apreciarse, el aspecto más difícil del mapeo es la obtención de nueva información del modelo por encima definir una plantilla con mucha información en ella. Esta será la guía para todas las mejoras propuestas. La mejora propuesta es primero reemplazar el anidamiento de estructuras con nombre por las estructuras anónimas de ISO C11 que vimos en Ben Klemens (pero en este caso para la tabla virtual), con esto los métodos de la tabla virtual pueden inicializarse con su nombre:

```
static ChildVtbl const vtbl={
.method1 = Parent_method1_
.method2 = Child_method2_
}
```

Luego para poder mantener un listado de las últimas implementaciones, una macro de inicializaciones puede mantenerse, el siguiente sería el template de la misma:

```
#define [aClass.name.toUpper()/]_METHOD_IMPLEMENTATIONS \
    [aClass.superClass.toUpper()/]_METHOD_IMPLEMENTATIONS \
    [for (o : Operation |  aClass.allNotFinalOperations() )]
    .[o.name/] =  [aClass.name/]_[o.name/]_,
    [/for]
```

Luego de esta plantilla simple de generar, una plantilla para la instanciación de la tabla virtual se reduciría a:

```
static [aClass.name/]Vtbl const vtbl={
[aClass.name.toUpper()/]_METHOD_IMPLEMENTATIONS
}
```

Esto es posible ya que desde C99 la asignación utilizada en la lista de asignaciones es la última que aparece. De esta manera nos hemos deshecho de todas las consultas que necesitamos para la instanciación de la tabla virtual.

Una dificultad adicional a resolver sería no agregar a la declaración de la tabla virtual métodos redefinidos polimórficamente sin tener que consultar con la consulta `getNotRedefine ()`.
Este se puede lograr creando macros y compilaciones condicionales por cada método polimórfico:

```
typedef struct [aClass.name/]Vtbl={
    struct [aClass.superClass.name/]Vtbl;
        [for (o : Operation |  aClass.allNotFinalOperations()
↪   )]
        #ifndef [o.name.toUpper()/]
        #define [o.name.toUpper()/]
        [o.generateSOOPCMethodPrototype()/],
        #endif
        [/for]
}
```

Ahora una clase hija que defina de nuevo el método polimórfico no lo estará incluyendo en su estructura de tabla virtual por más que desconozca que

se trata de una redefinición. La desventaja de esto es que ahora no podrán existir dos métodos con el mismo nombre. De esta manera también pasamos la resolución de esta problemática al precompilador C.

6.5.2 Plantilla para el prototipo de un método polimórfico en SOOPC

Los métodos polimórficos son declarados por primera vez en alguna clase tabla virtual de alguna clase con el primer argumento (`self`) siendo del tipo de esa clase. Si una clase heredada quiere redefinir dicho método, deberá respetar su firma original para no obtener errores de parseo. Para esto es necesario poder saber cual es la primer clase que definió el método. Esto se puede conseguir con una consulta `getFirstClassDefinerName()` que funcione a la inversa de la ya presentada `getLastClassDefinerName()`.

```
[template public getFirstClassDefinerName(classes:
↪  OrderedSet(Class) ,aMethod: Operation)
post(substituteAll('\t','').trim())]
[let c: uml::Class = classes->first() ]
[if (c.ownedOperation->exists(m:Operation | m.redefine(
↪  aMethod ) ))]
    [c.name/]
[else]
[comment] **Note** : at the time of writing, the OCL standard
↪  library sports a bug
which changes *OrderedSets* in *Sets* when excluding
↪  elements.[/comment]
    [getFirstClassDefinerName(classes -> asSequence() ->
↪  excluding(c) -> asOrderedSet() , aMethod)/]
[/if]
    [/let]
[/template]
```

Con esa consulta podemos generar la siguiente plantilla para los prototipos

de métodos virtuales:

```
[template public genVirtualMethodPrototype (o : Operation) ]
[generateReturnType(o)/] [o.class.getName()/]_[o.getName()/]_
↪  ([o.class.getFirstClassDefinerName(o)/] *
↪  self[for(p:Parameter | getOperationParameters(o))
↪  before(', ') separator(', ')][p.genParameter()/][/for])
[/template]
```

6.5.2.1 Verificación

Utilizando el modelo de la sección anterior, para el método `vPrint()` de Child la plantilla `genVirtualMethodPrototype()` genera el siguiente código.

```
void Child_vPrint_ (Parent * self)
```

Se puede ver que a pesar de que `Child` es la clase que está redefiniendo el método, el argumento utilizado para el argumento `self` es la clase `Parent` la cuál es la primera en declarar el método `vPrint()`.

6.5.2.2 Mejoras

De nuevo para simplificar el mapeo esta información puede proveerla el preprocesador de C. Si cuando declaramos el método definimos una macro con el nombre de la clase. Junto con la mejora anterior esto quedaría de la siguiente manera:

```
typedef struct [aClass.name/]Vtbl={
    struct [aClass.superClass.name/]Vtbl;
        [for (o : Operation |  aClass.allNotFinalOperations()
↪  )]
        #ifndef [o.name.toUpper()/]
        #define [o.name.toUpper()/]
```

```
        #define [o.name.toUpper()/]_SELF_TYPE [aClass.name/]
        [o.generateSOOPCMethodPrototype()/],
        #endif
        [/for]
}
```

Con esto definir el prototipo de la implementación de un método virtual solo requiere conocer el nombre del método, lo que es accesible desde la clase misma, para poder utilizar `[o.name.toUpper()/]_SELF_TYPE` como tipo del primer argumento.

6.5.3 Redefinición de métodos polimórficos en OOPC

Como vimos, OOPC soporta herencia múltiple. El framework se vale de una importante biblioteca de macros. Para poder redefinir un método se utilizan las macros `methodOvldDecl()` (o su equivalente para métodos constantes) y `methodOvldName()`. Ambas reciben como primer argumento el nombre del método y como segundo la clase que definió el método. Además se debe utilizar la macro `overload()` con toda la secuencia línea de ascendencia de clases hasta llegar a la que definió el método por primera vez. Esta tarea se complica aún más cuando 2 (o más) clases contienen un método con el mismo nombre y firma pero no pertenecen a la misma jerarquía de clases, si una clase hereda de estas 2 clases y define dicho método entonces deberá generar dos implementaciones y redefiniciones para cada clase método.

Primero intentaremos obtener distintos listados de de líneas de ascendencias que llegan hasta las clases que definen por primera vez al método.

El primer objetivo entonces es encontrar las clases que definen por primera vez el método. Para eso de nuevo debemos recurrir a la recursividad con la siguiente consulta:

```
[query
public
```

```
getFirstClassesDefinerName(aClass: Class ,anOperation:
↪  Operation): Set(Class)=

if (not aClass.superClass->isEmpty() and aClass.superClass
        ->exists( s:Class| s.hasOperation(anOperation))) then
    Set(Class){}->union ( aClass.superClass->select( s|
     s.hasOperation(anOperation))
        .getFirstClassesDefinerName(anOperation)->asSet() )
else
   Set(Class){}->including(aClass)
endif
/]
```

la idea es que una clase solo debe agregarse como definidora si las clases padre de la misma no contienen dicho método. Para preguntar si las clases padre contienen dicha operación utilizamos la consulta `hasOperation()`:

```
[query
public
hasOperation(aClass: Class ,anOperation: Operation): Boolean=
if( aClass.ownedOperation->exists( op
        | op.redefines(anOperation) ) ) then
    true
else
    if aClass.superClass->isEmpty() then
     false
    else
     aClass.superClass->exists(s:Class
        | s.hasOperation(anOperation))
    endif
endif
/]
```

La consulta `redefines()` debe ser cierta si el nombre de las operaciones y los atributos de sus argumentos (tipo, dirección, etc) en orden son iguales.

Esto lo podemos conseguir con las siguientes queries:

```
[query public redefines(op1: Operation,op2: Operation):
↪  Boolean=
if(op1.name = op2.name)then
 op1.ownedParameter->asOrderedSet()->forAll(
↪  par1:uml::Parameter |
 op2.ownedParameter->exists(par2:uml::Parameter |
↪  par2.type.name =
 par1.type.name and par1.direction = par2.direction and
 (par1.direction = ParameterDirectionKind::return
 or parameterIndex(op1,par1) = parameterIndex(op2,par2))))
else
 false
endif
/]

[query
public
parameterIndex (op1 : Operation,par1: Parameter) : Integer =
 op1.ownedParameter->asOrderedSet()->excluding
  (op1.ownedParameter->any(direction =
↪  ParameterDirectionKind::return))
   ->asOrderedSet()->indexOf(par1)/]
```

La siguiente tarea será conseguir todo el camino de herencia desde las clases que definen por primera vez la operación hasta la clase a generar. Si nos encontramos en un caso del problema del diamante los caminos pueden ser varios. Este problema es difícil de resolver en OCL y requerimos ayuda de la comunidad de Acceleo para resolverla.
Las siguiente consulta nos devuelve toda la jerarquía de clases:

```
[query public superPaths(aClass: uml::Class) :
↪  OrderedSet(Sequence(uml::Class))=
```

```
let superSuperPaths : Set(Sequence(uml::Class)) = if
↪  aClass.superClass->isEmpty()
then OrderedSet(Sequence(Class)){Sequence(Class){}}
else aClass.superClass->iterate(superClass1;
result : Set(Sequence(uml::Class)) =
↪  Set(Sequence(uml::Class)){}
| result->union(superPaths(superClass1)))
endif in
superSuperPaths->collectNested(superPath :
↪  Sequence(uml::Class) |
     superPath->prepend(aClass))->asOrderedSet()
/]
```

Con esta consulta podemos armar otra que nos devuelva todos los caminos de una clase a otra:

```
[query public getPaths(fromClass: uml::Class, toClass:
↪  uml::Class) : OrderedSet(Sequence(uml::Class))=
fromClass.superPaths()->select(path: Sequence(uml::Class) |
↪  path->includes(toClass))
->collectNested(path: Sequence(uml::Class) |
↪  path->subSequence(2,path->indexOf(toClass)))
->asOrderedSet()
/]
```

Debido a la dificultad que significó obtener esta consulta en OCL, incursionamos en cómo realizar la consulta en JAVA utilizando un envolvedor (wrapper en inglés) que facilita Acceleo.
El siguiente es el código del envolvedor:

```
[query public getPathsJAVA(arg0 : Class, arg1 : Class) : Set (
↪  Sequence (Class))
=
↪  invoke('org.eclipse.acceleo.module.oopc.services.UML2OOPCServices',
↪
```

```
'getPathsJAVA(org.eclipse.uml2.uml.Class,
↪  org.eclipse.uml2.uml.Class)',
Sequence{arg0, arg1})
/]
```

El mismo invoca al método referenciado dentro del método `invoke()`.
El código JAVA que resuelve la consulta es el siguiente:

```
import org.eclipse.uml2.uml.Class;

public class UML2OOPCServices {
    public Set<List<Class>> getPathsJAVA(Class fromClass, Class
    ↪  toClass) {

        Set<List<Class>> result = new HashSet<List<Class>>();

        if (fromClass == toClass)return result;

       EList<Class> superClasses = fromClass.getSuperClasses();

        if(superClasses.isEmpty())return result;

        for (Class superClass : superClasses) {
            Set<List<Class>> subPath = getPathsJAVA(superClass,
            ↪  toClass);
            if (subPath.isEmpty() && superClass==toClass){
                ArrayList<Class> justTheClass = new
                ↪  ArrayList<Class>();
                justTheClass.add(toClass);
                result.add(justTheClass);
            }
            else{
                for(List<Class> path : subPath){
                    path.add(0, superClass);
                    result.add(path);
```

```
                }
            }
        }
        return result;
    }
}
```

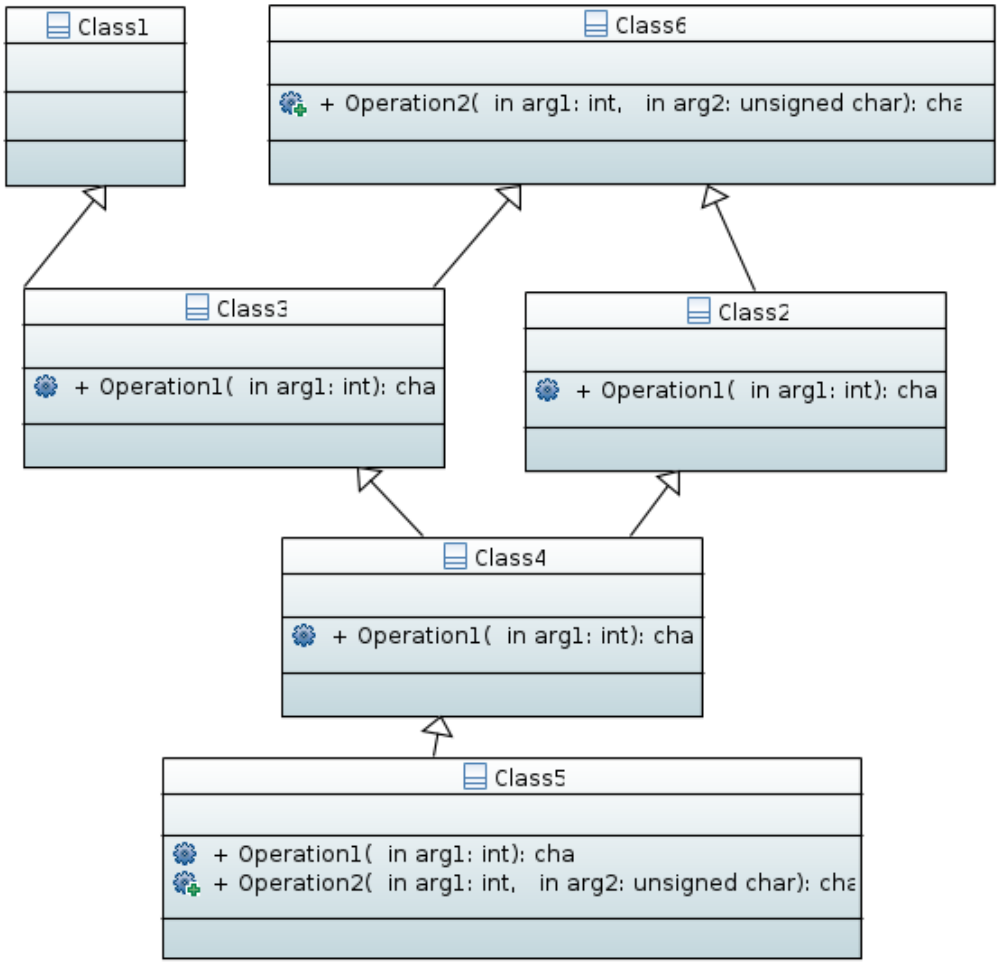
Fue mucho más sencillo resolver la consulta con JAVA ya que es un lenguaje conocido y gracias a las herramientas de depuración.

6.5.3.1 Mejoras

La mejora que proponemos para simplificar el mapeo es el uso de estructuras anónimas de ISO C11 en la forma en que los usa la especificación de Ben Klemens. Esto permitiría poder referenciar los métodos desconociendo la jerarquía de clases desde la clase a generar hasta dicha clase. Por otro lado, tendría como consecuencia no poder heredar más de dos clases de distinta jerarquía con métodos del mismo nombre.
La necesidad de conocer el nombre de la clase que declaró por primera vez el método, como vimos en SOOPC, tiene que ver con mantener la firma del mismo, esto se puede cambiar como propusimos allí con una macro y compilación condicional.

6.5.3.2 Verificación

El siguiente modelo se utilizó para la verificación de las consultas anteriores



Definimos la siguiente plantilla para utilizarla con la consulta `getFirstClassesDefinerName`

```
[template public genClass (aClass : uml::Class) ]
[for (anOp:uml::Operation | aClass.ownedOperation)]
    [for (aClass1 : uml::Class |
↪   aClass.getFirstClassesDefinerName(anOp))]
    [if(aClass1=aClass)]
    defmethod([aClass1.name/]_[anOp.name/])
    [else]
        [for (path:Sequence(Class) | getPaths(aClass,aClass1))]
```

```
    overload([for(c:Class|path)][c.name/].[/for][anOp.name/])
    ↪  = methodOvldName([anOp.name/],[aClass1.name/])
    [/for]

    [/if]
    [/for]

[/for]
[/template]
```

De esta manera para `Class5` el código generado es:

```
    overload(Class4.Class2.Operation1) =
↪   methodOvldName(Operation1,Class2)
    overload(Class4.Class3.Operation1) =
↪   methodOvldName(Operation1,Class3)
    overload(Class4.Class2.Class6.Operation2) =
↪   methodOvldName(Operation2,Class6)
    overload(Class4.Class3.Class6.Operation2) =
↪   methodOvldName(Operation2,Class6)
```

Si distintas clases definen por primera vez el método como en el caso del método `Operation1()` por las clases `Class2` y `Class3` entonces pueden obtenerse los caminos hacia cada una.

6.5.4 Declaración de métodos polimórficos en OOPC

La problemática es la misma que para SOOPC, los métodos polimórficos definidos por primera vez deben ser los únicos a ser incluidos en la tabla virtual. Las consultas presentadas para SOOPC servirán para OOPC también, principalmente porque `getEntireClassesHierarchechy()` puede utilizarse para devolver el set de clases padre incluso en un caso de herencia múltiple. Otra manera de realizar esta consulta es preguntando

si la consulta `getFirstClassesDefinerName()` introducida en la sección anterior devuelve la clase que se está generando como la primera en definir el método. Esta última manera fue la utilizada en la plantilla de la verificación anterior.

6.5.4.1 Mejoras

La mejora propuesta sigue siendo en todos los casos mediante el uso de macros y compilación condicional.

6.5.4.2 Verificación

El modelo y la plantilla anterior servirán de verificación. Para `Class2` el código generado es:

```
defmethod(Class2_Operation1)
```

con lo que se indica que el método `Operation1()` es la primera vez que se define.

6.5.5 Declaración de métodos polimórficos en OOC de Tibor Miseta, OOC-S y GObject

De nuevo para saber si declarar un método polimórfico en la tabla virtual o no debemos saber si el mismo no se trata de una redefinición de método. Este caso es distinto a los anteriores ya que un método no sólo pudo haber sido definido en una clase padre sino también en una interfaz o mixin que estemos realizando o que haya sido realizada por una clase padre. Para eso podemos excluir dichos métodos polimórficos utilizando la consulta `getNotInterfaceRedefiner()`

```
[query
public
```

```
getNotInterfaceRedefiner (seq : Sequence(Operation) )
  : Sequence(Operation) =
 seq->select(op : uml::Operation
  | not(op.isAnInterfaceMethodRedefinition()))
 ->asSequence()/]


[query
public
isAnInterfaceMethodRedefinition (anOperation : Operation ) :
↪  Boolean =

↪  anOperation.class.getEntireClassesHierarchechy().getAllInterfaces()
  .getOwnedOperations()->exists(anInterfaceOperation:Operation
  |redefines(anInterfaceOperation,anOperation))/]
```

La forma de obtener las interfaces de una clase es a través de la relación de realización como en la siguiente consulta:

```
[query public getAllInterfaces(aClass : Class) :
↪  Bag(Interface) =
aClass.clientDependency->filter(Realization).supplier
 ->filter(uml::Interface)
/]
```

La siguiente plantilla genera la declaración de métodos nuevos para una clase escrita bajo OOC de Tibor Miseta:

```
Virtuals( [aClass.name/], [aClass.getSuperClassName()/] );


[let seqOp : Sequence(uml::Operation) =
↪  aClass.getOwnedOperations()->getVirtual()->

↪  getNotInline()->getNotRedefine()->getNotInterfaceRedefiner()]
```

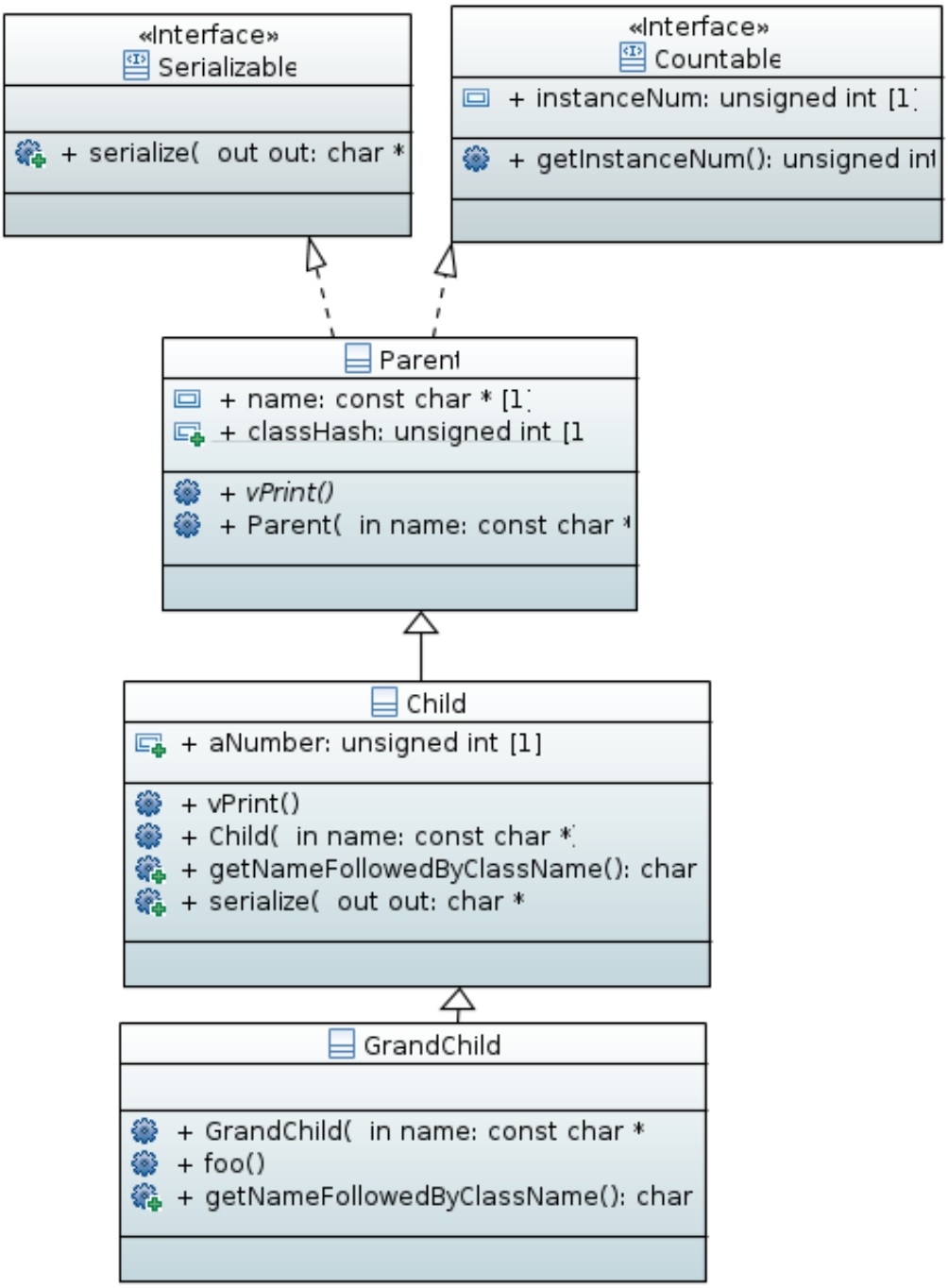
```
[for (o : Operation | seqOp)  ]
    [o.generateReturnType()/]
↪   (*[o.name/])([o.generateOOCMethodArguments()/]);
[/for]
[/let]

[for (namedElement : NamedElement | aClass.getAllInterfaces())]
    Interface( [namedElement.name/] );
[/for]

EndOfVirtuals;
```

6.5.5.1 Mejoras

Al igual que para el caso de la redefinición de métodos definidos en clases, la solución para esto es mediante una macro y compilación condicional como en SOOPC. Eso permitiría generar la operación sin saber si es que pertenece a una interfaz o no dejando la tarea al precompilador. En las interfaces sólo se debe generar la macro para cada método sin agregar la compilación condicional ya que las interfaces en estos dos frameworks no pueden heredar de otras. Esta solución no es aplicable en el caso de OOC-S ya que los métodos están definidos dentro de una macro. Otra solución podría venir por el lado del modelo especificando, mediante un estereotipo, si se trata de una definición de un nuevo método.

6.5.5.2 Verificación

Utilizamos el siguiente modelo para verificar las consultas



Para `Child` se generó la siguiente definición de nuevos métodos polimórficos:

```
Virtuals( Child, Parent );
    char * (*getNameFollowedByClassName)(Child self);
EndOfVirtuals;
```

Como puede apreciarse, a pesar de que Child define un método polimórfico `serialize()` el mismo es identificado como declarado por primera vez por una interfaz realizada por una clase padre por lo que la misma no es incluida.

6.5.6 Inicialización de tabla virtual en OOC de Tibor Miseta y GObject

Como en los frameworks anteriores que generan sus tablas virtuales anidando estructuras no anónimas (no como en el caso de Ben Klemens o OOC-S) para la inicialización de la tabla virtual se debe conocer la clase que realiza por primera vez cada método. En este caso además, cuando el método pertenece a una interfaz, se debe saber cuál es la clase que realiza por primera vez la interfaz. La siguiente plantilla inicializa la tabla virtual dentro de la función de inicialización para OOC (para GObject se precisan algunas modificaciones menores):

```
[let aClassVirtuals : Sequence(Operation)
    = aClass.getOwnedOperations()->getVirtual()]
[for (classLink : Class
 | getEntireClassesHierarchechy(aClass)->reverse()) ]
  [let classLinkVirtuals : Sequence(uml::Operation)
    = classLink.getOwnedOperations()
    ->getNotRedefine()->getNotInterfaceRedefiner() ]
     [for (o : Operation
    |  aClassVirtuals->intersection( classLinkVirtuals ) ) ]
(([classLink.getName()/]Vtable)vtab)->[o.name/] =

↪   [o.generateMethodUpCast(classLink)/][o.class.name/]_[o.name/];
     [/for]
  [/let]
```

```
  [for ( aClassLinkInterface : uml::Interface
    |classLink.getAllInterfaces().oclAsType(uml::Interface) )]
      [let aClassLinkInterfaceOperations :
↪   Sequence(uml::Operation)
    = aClassLinkInterface.getOwnedOperations() ]
     [for (anInterfaceOperation : Operation
      |
↪   aClassVirtuals->intersection(aClassLinkInterfaceOperations))]
(([classLink.getName()/]Vtable)vtab)-> \
[aClassLinkInterface.getName()/].[anInterfaceOperation.getName()/]
↪   = \
[anInterfaceOperation.generateInterfaceMethodUpCast(aClassLinkInterface)/]
↪   [aClass.getName()/]_[anInterfaceOperation.getName()/];
     [/for]
      [/let]
  [/for]
[/for]
[/let]
```

En este caso el framework exige un casteo de la implementación de los métodos para asignarlos a la tabla virtual. Esto también requiere conocer la clase o interfaz que lo definió por primera vez.

6.5.6.1 Mejoras

De nuevo la mejora propuesta será anidar estructuras anónimas de ISO C11. Para el casteo de los métodos propondremos crear una tipo junto al método en la tabla virtual que posea la firma del método.

6.5.6.2 Verificación

Para el modelo presentado en la sección anterior se generó la siguiente inicialización de tabla virtual para la clase Child:

```
ChildVtable vtab = & ChildVtableInstance;

((ParentVtable)vtab)->vPrint = (void(*)(Parent
↪  self))Child_vPrint;
((ParentVtable)vtab)->Serializable.serialize =
        (void(*)(Object self, char * out))Child_serialize;
((ChildVtable)vtab)->getNameFollowedByClassName =
        (char *(*)(Child
↪  self))Child_getNameFollowedByClassName;
```

6.5.7 Constructor de MetaClase en OOC de Axel Tobías Schreiner

Para este framework habíamos presentado dos propuestas de expresión, una con metaclases implícitas y otra con explícitas. El código desde un modelo con metaclases explícitas solo requiere de información directa para sus plantillas. En el caso de un modelo con metaclases implícitas se debe saber si los métodos polimórficos declarados por la clase corresponden a una redefinición o no, esto último implica que una nueva metaclase debe declararse y los métodos nuevos se deben declarar en su constructor. Esto se puede lograr a través de la consulta `getFirstClassesDefinerName()` que puede compararse para saber si es igual a la clase actual.

Esta observación de la diferencia de dificultad en el mapeo al código entre dos formas de modelado puede ser útil a la hora de implementar cualquier generador de código desde UML.

6.6 Dificultades encontradas

Acceleo y OCL son lenguajes muy declarativos, es decir basados en la matemática más que en la manera de razonar de las personas, esto nos ha dificultado la resolución de los mapeos. Como alternativa a las consultas en OCL, JAVA puede utilizarse el cual es un lenguaje mucho más amigable para la mayoría de los programadores. La única dificultad en realizar consultas

en JAVA es tener que codificar un envolvedor (wrapper en inglés) para cada una. Sería importante generar un estudio comparativo entre Acceleo y otras tecnologías para generar código desde UML, Xtend por ejemplo está siendo ampliamente usado en distintos proyectos como Yakindu[1] y Papyrus.[2]

Una desventaja de acceleo actualmente es la falta de un framework de testing, aunque puede ser minimizado por la variedad de herramientas de desarrollo para OCL, el lenguaje de las consultas en acceleo. Las consultas escritas en JAVA sí pueden ser testeadas, por ejemplo, con JUnit.[3]
Si bien se decidió codificar tres generadores para UML generators, no se obtuvo un impulso por parte de su comunidad ya que desde el comienzo de esta tesis la misma declinó rápidamente, los últimos plugins publicados del proyecto son para Eclipse Mars (al momento de escritura de esta tesis 3 versiones atrás de la última versión). Por el contrario, la comunidad de Acceleo sí es una comunidad activa.

6.7 Resumen

Hemos podido escribir un generador de código completo para algunos de los frameworks analizados y hemos analizado los puntos necesarios para hacer factible generar código para los seis frameworks objetivo. Para eso elegimos Acceleo dentro del conjunto de herramientas especializadas en generación de código desde UML. La necesidad de información adicional, comparado con otros LPOO, que poseen los frameworks nos llevó a escribir consultas en OCL (tecnología central en Acceleo), un lenguaje altamente declarativo y que hemos encontrado difícil de usar, la dificultad en una de las consultas nos llevó a incursionar en las consultas escritas en JAVA, las mismas resultan una herramienta muy conveniente. Esta necesidad de información adicional podría suplirse modificando la especificación de codificación de los frameworks, esto puede traer consecuencias importantes como tener que cambiar el estándar en el que se está escribiendo el código, la consecuencia de tener que escribir más líneas de código es relativa si se está utilizando

[1] https://www.itemis.com/en/yakindu/
[2] https://www.papyrus.com/
[3] https://junit.org/

un generador de código. Otra manera de simplificar el mapeo es utilizando una forma de modelar que requiera inferir menos información (como vimos para para OOC en metaclases explícitas respecto de las implícitas).
Hemos utilizado el conocimiento adquirido para esta tesis para implementar un generador de código para los frameworks SOOPC, OOC de Miseta y Dynace. Para esto hemos modificado el proyecto UML Generators y en especial el generador UML to Embedded C Generator. El proyecto resultante puede encontrarse en el repositorio especificado en el apéndice. Hemos tenido la oportunidad de utilizar el generador en los procesos de producción de una empresa dedicada a los sistemas embebidos de forma satisfactoria.

Capítulo 7

Conclusiones y trabajo futuro

Tres han sido los ejes que han impulsado el desarrollo de esta tesis.
1. ¿Qué clase de diagramas de clase UML son fácilmente mapeables a código C escrito bajo un framework de orientación a objetos tomando en cuenta sus conceptos soportados?
2. ¿En qué medida es más sencillo expresar bajo estos frameworks artefactos como clases o interfaces comparado con hacerlo desde el código?
3. ¿Cuál es la factibilidad y sencillez de implementar un generador de código a partir de los diagramas?

Las conclusiones que hemos podido sacar en cada punto son:
1. La cantidad de frameworks de OO escritos en C son muchos y muy variados en sus conceptos soportados en gran medida por sus diferencias en sus modelos de objetos. Este soporte de conceptos permite la utilización, en general intuitiva, de diagramas de clase que los expresan y han quedado ejemplificados en este trabajo. Escogimos analizar nueve frameworks. Sólo seis de ellos resultaron convenientes para su aplicación en sistemas altamente restringidos (con menos de 1MB de memoria), respecto de los demás precisan más recursos de memoria o hasta un SO embebido (GObject). La gran variedad de conceptos soportados entre todos ellos nos lleva a una conclusión en el área de la **educación**: podemos enseñar prácticamente todos los conceptos que han estado relacionados a la orientación a objetos sin tener que salirnos de un único lenguaje (por demás sencillo y que muchas veces

es el introductorio en muchas carreras relacionadas al software); además el costo de la programación bajo estos conceptos es mucho más visible.
2. Salvo para los frameworks que utilizan una herramienta externa en el proceso de convertir código fuente en artefactos como bibliotecas o ejecutables, las dificultades en el prototipado de clases y su evolución escribiendo nuevos métodos son variadas e importantes comparadas con otros LPOO, se utiliza mucho código de plantilla, se repite información en el código, se empeora la legibilidad del código si la implementación de conceptos está mezclada con la implementación de la solución al problema y se debe tener conocimiento de la jerarquía de clases o por lo menos de si el método polimórfico declarado se trata de una redefinición o no. Esto empeora la experiencia del programador al tener que estar concentrado en la implementación de conceptos de OO en vez de la solución del problema y al obligarlo a escribir más código con posibilidad a equivocarse en el proceso. UML ha sido concebido para expresar con sencillez los conceptos de la orientación a objetos por lo que no son objeto de las dificultades anteriores. Otra forma de expresión utilizada para disminuir la dificultad es la de un preprocesador externo, el mismo implica dejar de codificar en C por lo que muchas herramientas como las de análisis de código, refactoring y autocompletar se vuelven obsoletas, además un nuevo lenguaje no estándar debe aprenderse. UML es un lenguaje estándar y no reemplaza el lenguaje de implementación que es usado.
3. Hemos podido escribir un generador de código completo para algunos de los frameworks analizados, el código del mismo se encuentra en el repositorio especificado en el apéndice y hemos analizado los puntos necesarios para hacer factible generar código para los seis frameworks objetivo. Para eso elegimos Acceleo dentro del conjunto de herramientas especializadas en generación de código desde UML. La necesidad de información adicional, comparado con otros LPOO, que poseen los frameworks nos llevó a escribir consultas en OCL (tecnología central en Acceleo), un lenguaje altamente declarativo y que hemos encontrado difícil de usar, esto nos llevó a incursionar en realizar consultas en JAVA en vez de en OCL lo que ha resultado muy conveniente cuando la dificultad de la consulta se vuelve grande. La necesidad de información adicional podría suplirse modificando la especificación de codificación de los frameworks, esto puede

traer consecuencias importantes como tener que cambiar el estándar en el que se está escribiendo el código, la consecuencia de tener que escribir más líneas de código es relativa si se está utilizando un generador de código. Otra manera de simplificar el mapeo es utilizando una forma de modelar que requiera inferir menos información (como vimos para para OOC en metaclases explícitas respecto de las implícitas). El resultado de tener un generador de código en UML ha mostrado ser muy bueno en el área de **desarrollo**, además de las ventajas inherentes para cualquier lenguaje de programación, mejora la experiencia del programador COO volviéndola parecida a trabajar con otros LPOO y que es la forma adecuada de compararla con ellos: desde el toolsuite utilizado y no desde el lenguaje en aislamiento. El problema en la legibilidad del código se resuelve generando un archivo aparte para la implementación de los métodos que no se mezcla con la implementación de la OO en lenguaje C. Para la **educación** si se desea enseñar orientación a objetos con estos frameworks un generador de código desde UML puede facilitar la tarea.

7.1 Trabajo futuro

Nuestra investigación en COO y UML y nuestra experiencia en la generación de código COO desde UML nos lleva a marcar la importancia del siguiente trabajo futuro:

1. La propuesta de esta tesis fue utilizar un generador de código UML como facilitador de la codificación de COO sin tener que recurrir a un preprocesamiento no estándar. COS ha mostrado ser un framework que permite escribir código C estándar sin las dificultades en codificación que ello presenta, para esto se vale de una increíble biblioteca de macros C99 y de un paso de recolección de símbolos y relinkeo. COS no es aplicable a sistemas altamente restringidos pero puede ser objeto de estudio para producir nuevos frameworks que sí lo sean y no posean las dificultades de codificación antedichas.

2. Cómo modelar los multimétodos (soportados por COS) en UML requiere un análisis profundo ya que fueron concebidos luego de que UML haya sido

creado y puede no ser un buen lenguaje de modelado para los mismos. Esta tesis puede ser tomada como puntapié para el modelado de estos pero un estudio más formal es requerido.

3. Al haber logrado satisfactoriamente crear generador de código desde UML, el costo de líneas de código generadas es despreciable por lo que nuevos frameworks pueden surgir explotando esta característica, por ejemplo para aumentar las facilidades al usuario de clases.

4. Utilizar Acceleo como tecnología para la generación de código ha mostrado sus contras en un contexto en que se debe extraer información indirecta al diagrama. Podría analizarse si bajo otras tecnologías como Xtend lo hacen más simple.

5. Nuestro estudio explotó el lado de la ingeniería directa, pero se vería sumamente potenciado si se desarrollara también la ingeniería inversa, permitiendo expresarse en código o en modelos según le resulte más cómodo al programador en cada caso.

6. Los modelos permitidos por cada framework son reducidos en comparación a los que UML permite (por ejemplo en permitir herencia múltiple, esto no está limitado a ciertos frameworks sino que lenguajes como JAVA también tienen esas limitaciones). Mediante OCL se puede obtener un verificador de UML para constatar que los modelos se ajustan a los límites de cada framework.

Apéndice: Estructura del repositorio que acompaña este escrito

Ubicación

El código y los proyectos que acompañan a esta tesis y de los cuáles hemos hecho referencia en la misma se encuentran en github: https://github.com/jonyMarino/tesis así como en el CD que se entrega con esta tesis. Dentro del mismo se encuentran archivos `README` que indican cómo utilizar y compilar cada proyecto.

Estructura

La siguiente es la estructura simplificada de directorios del repositorio:

```
.
  ejemplos_de_codificacion_para_cada_framework
      BenKlemens
      COS
          COS
          COSDemo
      Dynace
          Dynace
```

```
        DynaceDemo
    GObject
    OOC_ATS
            ejemplo_tesis
    OOCS
    OOC_Tibor_Miseta
    OOPC
    SOOPC
generador_de_codigo_para_SOOPC_OOC_Y_DYNACE
    UML2ooec
        org.eclipse.umlgen.gen.embedded.c
            plugins
                org.eclipse.umlgen.gen.embedded.c
                org.eclipse.umlgen.gen.embedded.c.profile
    verificaciones_generacion_de_codigo_para_OOC_TM
    verificaciones_generacion_de_codigo_para_SOOPC
consultas_para_un_generador_de_codigo_para_OOPC
    org.eclipse.acceleo.module.oopc
        model
        src
        src-gen
```

Dentro de la carpeta `ejemplos_de_codificacion_para_cada_framework` se encuentra el código utilizado en la sección *Estado del arte en programación orientada a objetos en C* para analizar la codificación bajo cada framework. La misma contiene nueve carpetas correspondientes a los nueve frameworks. Dentro de la carpeta `Dynace` y `COS` se tiene un sub módulo de los repositorios de cada framework, el código de ejemplo para COS se encuentra en COSDemo y el de Dynace en DynaceDemo. Los ejemplos de SOOPC, OOC de Tibor Miseta y Dynace fueron generados con el generador de código desde UML a excepción del archivo main.c y el cuerpo de los métodos.
Dentro de la carpeta `generador_de_codigo_para_SOOPC_OOC_Y_DYNACE` tenemos un sub módulo con un clon del repositorio de *UML Generators* con las modificaciones realizadas al *UML to Embedded C generator* tanto en el generador como en el *profile* (donde definimos los estereotipos para

nuestro generador). También se encuentran en la carpeta dos proyectos utilizados para verificar el correcto funcionamiento del generador tanto para el framework SOOPC como OOC de Tibor Miseta, los mismos contienen un modelo UML modelado con papyrus y el código generado se encuentra en la carpeta `src-gen`.

Por último, la carpeta `consultas_para_un_generador_de_codigo_para_OOPC` contiene un proyecto Acceleo dónde se encuentran en la carpeta `src` las consultas necesarias para implementar un generador de código para el framework OOPC. En la carpeta `model` se encuentra el modelo utilizado para corroborar el correcto funcionamiento de las consultas. En la carpeta `src-gen` se encuentra el código generado por la plantilla que utiliza dichas consultas.

Referencias

Acceleo, 2019. Acceleo. Available at: {https://www.eclipse.org/acceleo/}.

Anón, 2019. Astah. Available at: {http://astah.net}.

Architect, E., 2017. Enterprise Architect. Available at: {http://bit.ly/2CTKorP}.

Barr, M., 2018. *Embedded C Coding Standard*, 20251 Century Blvd, Suite 330 Germantown, MD 20874: Barr Group.

Booch, G., 1991. *Object Oriented Design with Applications*, Benjamin/Cummings.

Deniau, L., 2007a. OOC2.0. Available at: {http://cern.ch/laurent.deniau/html/ooc-2.0.tgz}.

Deniau, L., 2007b. OOCS. Available at: {http://cern.ch/laurent.deniau/html/ooc-s.tgz}.

Deniau, L., 2001. OOPC. Available at: {http://ldeniau.web.cern.ch/ldeniau/html/oopc/oopc.html}.

Deniau, L., 2009. *The C Object System Using C as a High-Level Object-Oriented Language*, CERN – European Organization for Nuclear Research.

Douglass, B.P., 2010. *Design Patterns for Embedded Systems in C: An Embedded Software Engineering Toolkit*, Newnes.

Eckel, B., Strong Typing vs. Strong Testing. Available at: {http://bit.ly/2CSS6lS}.

Eclipse, 2016. UML Generators. Available at: {https://wiki.eclipse.org/Eclipse_UML_Generators}.

Estrada, D.B., 2007. Eliminando la Herencia Múltiple y el Diamante de la Muerte. Available at: {https://www.americati.com/doc/diamante/diamante.html}.

GLib, 2019. GObject. Available at: {https://developer.gnome.org/gobject/stable/}.

Google, 2019. Google Test. Available at: {https://github.com/google/googletest}.

Grady Booch, I.J., James Rumbaugh, 1998. *Unified Modeling Language User Guide, The*,

Grenning, J.W., 2011. *Test Driven Development for Embedded C*, Pragmatic

Programmers.

Hendrickx, S., 2004. *Glib-C: C as an alternative Object Oriented Environment*, UNIVERSITEIT ANTWERPEN.

IBM, 2010. Rational Rhapsody. Available at: {http://ibm.co/2QrkSSE}.

index, T.T.P.C., 2018. https://www.tiobe.com/tiobe-index/.

ITRS, 2011. The International Technology Roadmap for Semiconductors - Design. Available at: {http://www.itrs.net}.

Klemens, B., 2013. *21st Century C: C tips from the new school, 1st edition*, O'REILLY.

Lennis, L. & Aedo, J., 2013. *Generation of Efficient Embedded C Code from UML/MARTE Models*, Department of Electronic Engineering, University of Antioquia, ARTICA, Medellín, Colombia.

Madsen, O.L., 1988. *What object-oriented programming may be - and what it does not have to be*, Lecture Notes in Computer Science.

McBride, B., 2004. *Dynace*, Available at: {https://github.com/blakemcbride/Dynace}.

Miseta, T., 2017. *Object Oriented C (ooc) toolkit*,

M. Maranzana, J.L.S., J. -F. Ponsignon & Bernier, F., 2004. *Timing performances of automatically generated code using mda approaches*, ECSI.

Murillo, L.G., 2009. *Bridging the Gap Between Model Driven Engineering and HW/SW Codesign*, ALaRI Institute, Lugano, Switzerland.

OMG, 2017. *OMG Unified Modeling Language (OMG UML) Version 2.5.1*, OMG.

Peter W. Madany, P.K., Nayeem Islam, 1992. *Reification and reflection in C++: an operating systems perspective*, Technical Report UIUCDCS–R–92–1736,Department of Computer Science, Urbana-Champaign.

Samek, M., 2012. Economics 101: UML in Embedded Systems. Available at: {https://embeddedgurus.com/state-space/2012/04/economics-101-uml-in-embedded-systems}.

Samek, M., 2015. *Simple Object-Oriented Programming in C*, Quantum Leaps.

Schreiner, A.-T., 1993. *Object-oriented Programming with ANSI-C*,

Schreiner, A.T., 1993. *Object Oriented Programming with ANSI C*,

Sommerville, I., 1996. *Software Engineering.*, Addison Wesley.

Stroustrup, B., 1991. *What is «object-oriented programming»?: In G. Goos and*

J. Hartmanis, editors, Proc. European Conf. on Object-Oriented Programming, Paris (France), 1987, revisited 1991. Lecture Notes in Computer Science no. 276., Springer-Verlag.

XPand, 2019. XPand. Available at: {https://wiki.eclipse.org/Xpand}.

Xtend, 2016. Papyrus, Adding a New Code Generator. Available at: {http://bit.ly/37gEPBt}.