
**FACULTAD DE INGENIERIA**

**DIVISION DE ESTUDIOS DE POSTGRADO**

**MAESTRIA EN COMPUTACION**



# IMPLEMENTACION DE REFLEXION EN C++

**MERIDA, VENEZUELA**

**ENERO, 2009**


**FACULTAD DE INGENIERIA**

**DIVISION DE ESTUDIOS DE POSTGRADO**

**MAESTRIA EN COMPUTACION**

# IMPLEMENTACION DE REFLEXION EN C++



**Presentado por: MARCO ANTONIO ADARME JAIMES**

**Como requisito para obtener el título de Magíster en Computación**

**Tutor: Dr. LEANDRO LEON**

**MERIDA, VENEZUELA**

**ENERO, 2009**

**RESUMEN**

El propósito de la tesis es la implementación de un soporte que permita realizar reflexión computacional en un dominio local. El soporte contiene los mecanismos para realizar consultas e invocaciones en forma dinámica de cualquier estructura de clase que se presente.

El mecanismo implementado se denomino "SIRC" o sistema de introspección (consultas-observación) y realización (intercesión o invocación dinámica) para C++. El sistema usa la herramienta GCCXML que permite representar la información de la clase en un archivo XML para su posterior lectura y carga en memoria; lo que da la facultad de realizar las operaciones de consultas en un proceso de introspección. La invocación, se realizó a través de apuntadores a funciones/métodos y con carga dinámica que son almacenadas por el sistema usando metaprogramación (técnica que permite escribir programas a través de otros).

"SIRC" posee un conjunto de objetos que permite a través del nombre de la clase y su código fuente observar, conocer e invocar métodos y atributos dinámicamente.

Palabras claves: reflexión computacional en C++, introspección, RTTI, metaobjetos.

## ÍNDICE GENERAL

## ÍNDICE DE FIGURAS

## ÍNDICE DE TABLAS


Sistema de Introspección y Realización en C

# INTRODUCCION

La reflexión computacional es la capacidad que tiene un programa de examinarse y modificarse a sí mismo, así como de exportar una interfaz para su uso. Programas que no conocen a priori la composición de una clase pueden instanciar objetos implícitamente en el universo de su ejecución. Eventualmente, un objeto puede razonar y actuar sobre sí mismo y ajustar su comportamiento en función de ciertas condiciones [1]. Un sistema reflexivo es capaz de usar su propia representación para extender, modificar, analizar su computación [2] y así cambiar sus características en tiempo de compilación o de ejecución.

Desde el interés de la orientación a objetos, la reflexión es un mecanismo que consiste en ofrecer capacidad de exportar, dinámicamente, especificaciones de clases de objetos, junto con todas las especificaciones de sus métodos, así como instancias de objetos que pueden invocarse por referencias locales.



La reflexión puede tener dos comportamientos. El primero denominado "Introspección", es la capacidad de inspeccionar la estructura de un programa. En la orientación a objetos esta capacidad se ve reflejada en la operación de consultas de cada parte de una clase; es decir, de sus métodos y atributos, su tipo, modificadores de acceso, valores de retorno, entre otros. El segundo comportamiento, denominado "Realización" o "Intercesión", es la capacidad de extender la computación al punto de realizar "invocaciones" de los servicios de una clase en forma dinámica.

En la actualidad programas escritos en Java [49] [50] o C # tienen acceso a estos comportamientos a través de su API de reflexión, los cuales poseen un conjunto de objetos y métodos que definen la estructura del programa y permiten el análisis y modificación del estado de sus objetos; cosa distinta a C++ que no cuenta con un soporte nativo para reflexión.

El fin de este trabajo es realizar un soporte, que no subyazca en ninguna forma de compilación, de reflexión en introspección e invocaciones dinámicas de una clase en el lenguaje de programación C++ para plataformas GNU-Linux, usando el compilador g++ (gcc). Cuenta habida de la complejidad con el encadenamiento de código y la heterogeniedad de plataforma, este trabajo se restringe a la reflexión en un dominio local; es decir, a soportarla en un solo sitio a nivel inter e intra-proceso. En este sentido, se requieren mecanismos para:

- Especificar clases de objeto: al realizar esto, la clase queda "visible" al mundo exterior. El mecanismo exporta una interfaz o servicios que provee la clase. Obtiene información acerca de métodos y tipos con sus identificadores respectivos e interactúa con los métodos así como con sus valores de retorno.

- Relacionar una clase con el mecanismo de reflexión. Esta operación establece un enlace entre la interfaz de clase reflexiva y una clase de implantación.



- Instanciar objetos pertenecientes al dominio de reflexión, construir referencias hacia los mismos e invocarlos dinámicamente desde el mismo proceso donde reside el objeto o desde otro localizado en el mismo sitio.

El soporte de reflexión se desarrollo en un conjunto de clases agrupados en una librería que se denomino "SIRC" o Sistema de Introspección y Realización para C++. Para la operación de introspección se utilizó la herramienta denominada gccxml que permite convertir una clase a una representación XML y para el proceso de realización se utilizaron las librerías *Dynamic Loader Compatibility* (DLC) [33][34] que tienen el objetivo de cargar dinámicamente un función utilizando su código objeto en tiempo de ejecución.

El documento se estructura en cuatro capítulos. El esquema se organiza de modo que se parte de los aspectos más genéricos y se van concretando poco a poco desde la perspectiva del C++.

En el capítulo uno se presenta los conceptos relacionados con la reflexión computacional, arquitectura, tipos y su comportamiento en la programación orientada a objetos. Su fin es establecer una base adecuada para extraer las nociones que serán integradas en el diseño de "SIRC".

Los capítulos dos y tres proporcionan un análisis de la reflexión en el contexto del C++, se centra en estudiar las instrucciones nativas que ofrecen un acercamiento a las operaciones reflexivas básicas; de igual forma, se expone la descripción de las herramientas y soluciones que se han venido desarrollando en el área.

Finalmente, el capítulo cuatro que constituye el núcleo del trabajo. Tras considerar brevemente el problema de implementación de los mecanismos de introspección y realización; a partir de ésta, se presenta el análisis, diseño y construcción de "SIRC". Se describen las estrategias, herramientas y técnicas usadas para otorgar el soporte reflexivo que permitió la invocación dinámica de una clase.

# 1. VISION GENERAL Y CONCEPTOS RELACIONADOS CON LA REFLEXION COMPUTACIONAL

Un sistema computacional[1] se considera como algo que razona y actúa en relación con una parte del mundo llamado "dominio del sistema". Sus programas establecen la forma en que estos datos deben ser manipulados y establecen acciones que pueden o no realizarse para razonar y actuar de acuerdo al dominio [2].

Un programa está construido de líneas de código (también llamadas sentencias) en un determinado lenguaje de programación. La computación del sistema resulta gracias a la puesta en marcha del programa por un proceso de ejecución. La computación retorna resultados que llevan nueva información acerca del dominio o que actualiza su entorno (figura 1).



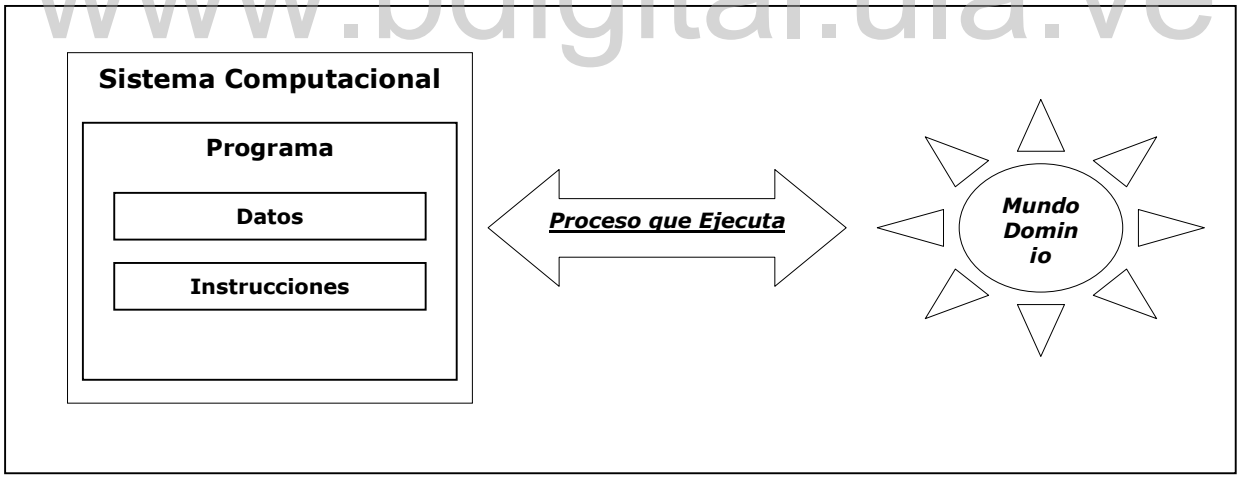



**Figura 1. Sistema Computacional**

Un sistema de información de matricula para una Universidad, por ejemplo, es un "sistema" cuyo dominio es el mundo que contiene a estudiantes, profesores, materias, facultades, departamentos; los datos corresponden a toda la información persistente en el sistema. El programa realiza algún proceso con estos datos

[1] En el desarrollo de la Tesis se irá a llamar "Sistema"

conocido como lógica del negocio[2], los manipula y ofrece información válida del dominio, según lo establecido a priori en un requerimiento del sistema.

El sistema ***razona*** **de manera muy estática** sobre su dominio cuando retorna resultados dependiendo de algún requerimiento funcional ya predefinido en un caso de uso (parte de Ingeniería de Software). El "razonar estático" implica que los sistemas convencionales sólo ofrecen funcionalidades ya programadas y adaptadas incluso a una plataforma de despliegue específica [2].

¿Qué sucede si el sistema se traslada de plataforma?, ¿Qué sucede si los requerimientos funcionales aumentan?, ¿Qué sucede si el SMBD[3] cambia?; la solución práctica depende la metodología de desarrollo que se use. En una metodología tradicional se tendría que cambiar gran parte del código para adaptarlo a estas necesidades; por otro lado, en metodologías de desarrollo a nivel de multicapa se operarían los cambios sobre las capas involucradas en la nueva implementación. Independientemente de la metodología siempre se ve la necesidad de alterar parte del código por medio del programador; es decir, que el nivel de adaptabilidad lo establece es quien manipula el código.



Por ejemplo, si El SMBD de un sistema de matricula cambia de Oracle a PostgreSQL, el "sistema" es capaz de reconocer el nuevo SMBD y busca el *driver* de conexión apropiado. Adicional a esto, reconoce a nivel de semántica que algunas consultas no concuerdan con el SQL nativo de la base de datos y altera las sentencias para que sean reconocidas por el SMBD. ¿Qué sucedió? , el sistema se adapto a los cambios de requisitos no funcionales sin necesidad de ayudarse de un programador; es decir el sistema "razonó" que necesitaba un nuevo *driver*, "razonó" que necesitaba cambiar su semántica en SQL y se adapto a los nuevos requerimientos. A estos tipos de sistemas se les denominan "sistemas reflexivos". Los sistemas reflexivos

[2] Son rutinas que realizan entradas de datos, consultas a los datos, generación de informes y más específicamente todo el procesamiento que se realiza detrás de la aplicación visible para el usuario; término usualmente dado en las metodologías de análisis y diseño orientado a objetos.

[3] Sistema Manejador de Bases de Datos

permiten razonar y afectar el programa (datos e instrucciones) para que puedan cumplir con los cambios dados en el dominio (figura 2).

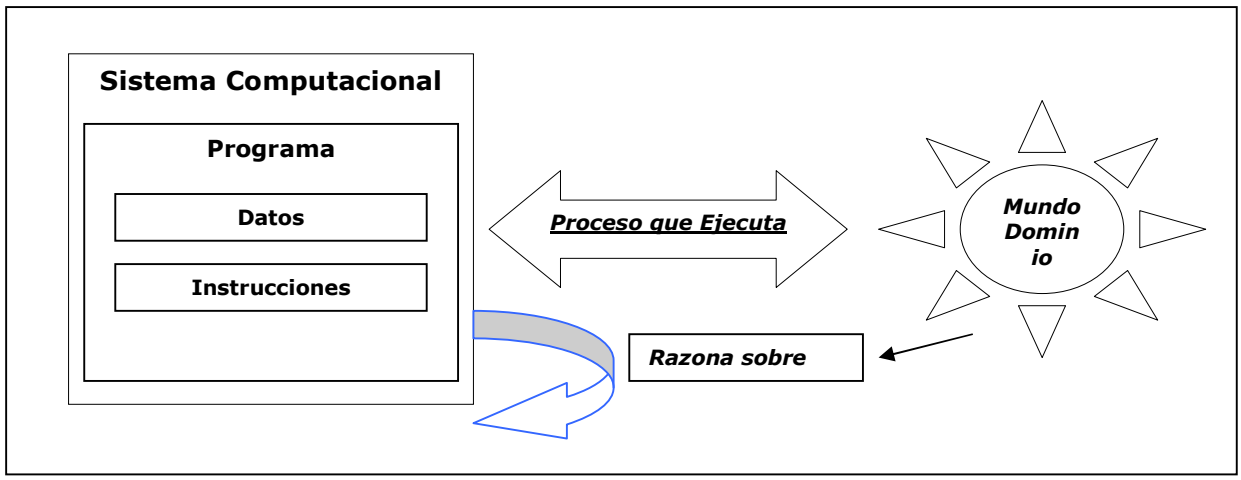



**Figura 2. Sistema Computacional "Reflexivo"**



## 1.1. REFLEXION COMPUTACIONAL

El concepto fue dado por Brian C. Smith [42] y fue tratado desde la perspectiva de la Inteligencia Artificial, donde se hace referencia a "sistemas que tratan y actúan sobre sí mismos". Con el referente dado por el Dr Smith se le une Pattie Mass, donde ofrece el concepto que hasta hoy se mantiene vigente:

> "Es la actividad desarrollada por un sistema computacional al realizar computaciones sobre y, posiblemente, afectando su propia computación" [1]

Que se concreta luego como:

> *"El comportamiento que exhibe un sistema reflexivo; donde un sistema reflexivo es un sistema computacional que trata sobre sí mismo, manteniendo una conexión causal"*

Aunque esta definición es muy general; se ubica ya en un contexto computacional y tiene varios puntos notables, donde se pueden destacar los términos *computación* y

*conexión causal*. La conexión causal se refiere a la relación que existe entre un sistema reflexivo y su estructura interna. Esta idea proporciona ya una primera aproximación del concepto, que hasta este momento era estrictamente autoreferencial y proporciona la base para todo un desarrollo posterior [2].

El término "*computación*" se refiere al tratamiento de la estructura de un programa fuente u objeto-binario (en el caso de C++). Esto lleva al estudio de procedimientos de identificación de las partes de un programa, el cual obedece de cierta manera a realizar un proceso de abstracción que ayude a determinar las características a nivel de datos, a nivel de instrucciones y demás partes integrales que lo componen.

La información según los niveles de abstracción se denomina metainformación[4]. La reflexión entonces estudia la forma como se obtienen e interactúan los datos de los diferentes niveles y los mecanismos que existen para "reflejar"[5] (computación en si mismo) esta información desde y hacia el exterior [7].



Lógicamente, la composición de un programa depende del paradigma de programación que se este usando; por ejemplo, en el caso de programas orientados a objetos la metainformación concierne a la identificación de clases, atributos y métodos.

Realizar "computación sobre sí mismo" implica que los programas conozcan y controlen su metainformación, en tiempo de compilación o en tiempo de ejecución [3] [7], en el desarrollo de este trabajo se irán presentando más ampliamente estos ítems y la complejidad que tienen a nivel de implementación en lenguajes que no soportan de manera nativa reflexión, como es el caso del C++.

[4] Metainformación: Se refiera a la representación de los componentes de un programa (clases, instrucciones, sentencias…) en entidades que describan su comportamiento y su semántica.

[5] Término usado en computación Reflexiva para representar todos los procedimientos que permiten obtener información de una clase así como de su manipulación (instanciación).

Esta definición resulta aún demasiado general; para comprender por completo cuáles son sus implicaciones, es necesario proporcionar una definición más técnica. Sin duda una buena definición fue elaborada por Jacques Malenfant [4], donde permite que el concepto sea fácilmente adaptado a los paradigmas actuales de programación e incluye el término de lenguajes reflexivos.

Malefant define reflexión como: "La habilidad integral de un programa para observar o cambiar su propio código, así como aspectos de su lenguaje de programación (sintaxis, semántica o implementación) [4], incluso en tiempo de ejecución. Se dice que un lenguaje de programación es reflexivo cuando sus programas tienen la habilidad antes descrita". La definición no se aparta de lo que inicialmente propone Smith; con una adición particularmente importante, pues involucra el lenguaje computacional como ente que debe proporcionar las estructuras para otorgar la habilidad antes mencionada. Así, se puede concluir que estos sistemas deben contar con mecanismos que permitan adaptar su semántica y procedimientos para que los programas puedan ser reflexivos y poder alterar dinámicamente su significado, término conocido como reificación [2][44].



Formalmente la reificación es definida por Malefant como:

"Es el proceso causal por el cual un programa de usuario P, o cualquier lenguaje L, que estaban implícitos en el programa traducido y en el sistema en ejecución, se hacen explícitos, utilizando una representación que se expresa en el propio lenguaje L y es puesta a disposición del programa P, que podrá inspeccionarla como si se tratase de datos ordinarios" [4].

En los lenguajes reflexivos, los datos obtenidos mediante la reificación están conectados causalmente con la información reflejada de modo que una modificación en uno de ellos afecta al otro. De esta forma, los datos obtenidos por la reificación son una representación fidedigna del correspondiente aspecto reificado. El objetivo primordial de la reificación es ofrecer mecanismos para que los datos se puedan obtener, consultar y ser modificados por el programa.

Con esta última definición, se puede decir que: "La reflexión computacional es el proceso por el cual un programa es capaz de acceder y alterar su propia estructura o la de otro programa en tiempo de compilación o tiempo de ejecución en un dominio local o distribuido". Esta funcionalidad permite añadir u obtener datos así como operaciones (funciones en términos de paradigmas imperativos) que tiene el programa a través de cadenas de caracteres que tienen correspondencia con los identificadores colocados en el código fuente. La reflexión también permite obtener e instanciar clases y métodos.

La reflexión se puede estudiar a nivel de dos (2) aspectos [2][4][5]: introspección o consultas sobre los componentes activos de un programa[6] (por ejemplo el nombre de un método, tamaño de un tipo de dato, entre otros) y la realización o la capacidad que tiene un programa para modificar dinámicamente su comportamiento, sin alterar su código inicial; por ejemplo, realizar invocaciones de servicios a través de cualquier mecanismo que el lenguaje tenga.



## 1.2. INTROSPECCION [8][9]

La introspección, como se ya mencionó, es la capacidad de un sistema para hacer consultas de si mismo. Este concepto puede variar dependiendo del lenguaje de programación que se este usando, por ejemplo:

- Para Java: es la habilidad que se tiene para manipular y examinar una clase. El lenguaje proporciona una API[7] que permite identificar clases y métodos, así como, la posibilidad de realizar instanciaciones.
- Para Perl: es la habilidad que se tiene para convertir un identificador en una variable y con ella acceder a su contenido.

[6] Se hace referencia a las partes de un programa como datos simples, estructuras de datos, clases-objetos y operaciones.

[7] Interfaz de programación de aplicaciones, es un conjunto de rutinas, procedimientos, protocolos, funciones y herramientas que una determinada biblioteca pone a disposición para que sean utilizados por otro software.

- En CORBA: se denomina a la operación de interceptar el llamado de un método y conocer el objeto que lo invoca. Este tipo de introspección no está basada en la estructura sino en comportamiento.

La definición siempre gira entorno al concepto de "consulta de un dato". En sistemas orientados a objetos la habilidad introspectiva permite obtener los nombres de las clases de los objetos, de atributos y métodos. Estas generan respuestas en cadena de caracteres que pueden ser pasadas a métodos que invocan otras operaciones de la forma:

**algunobjetoConIntrospección.obtenerMétodo("nombre_de_método);**

Con la introspección se puede realizar la construcción de componentes de una manera más flexible; por ejemplo, se podría descubrir las interfaces de un objeto y poder decidir cuál operación invocar. Con la facilidad del descubrimiento de las interfaces se tiene la posibilidad de interactuar con servicios que no necesariamente son definidos en el desarrollo original, característica vista en sistemas adaptativos [8].



Otra característica relevante que ofrece la introspección es la "Tolerancia a cambios", ya que, al cambiar algún servicio ofrecido por un objeto, por introspección se podría conocer las nuevas interfaces y comenzar a operar con ellas, reduciendo la tarea de modificar el código fuente de las aplicaciones lanzadoras (contenedoras). Los sistemas se vuelven tan flexibles que pueden reconfigurar sus componentes para adaptarlos al ambiente.

La introspección está disponible en diferentes sistemas los cuales ofrecen un sin número de información de sus partes [6]. Cada lenguaje con soporte reflexivo cuenta con los mecanismos necesarios para soportar reificación y la manera de almacenar su metainformación. Los tipos de información que se ofrecen con introspección son:

- **Identificación de tipos de datos:** comprende la operación de consulta de los tipos de datos. Permite conocer el tamaño y la clase de un objeto. Por ejemplo, lenguajes como Java existe una instrucción denominada "*instance of"* que permite en tiempo de ejecución conocer si un objeto es de una clase en particular.

- **Identificación estructural:** los sistemas presentan mecanismos para analizar la estructura de un programa dependiendo de la organización del código fuente. Para muchos sistemas OO [1][9], esto se realiza a través de mecanismos de descomposición que permiten ofrecer operaciones para identificar claramente que son clases, atributos, métodos y cuál es el cuerpo del programa.

- **Identificación de comportamiento:** así como se pueden identificar las partes de una clase en sistemas OO, también se puede consultar la información del objeto asociado la clase en tiempo de ejecución. Este tipo de información se determina identificando el comportamiento e interacciones que tiene el objeto con el programa que lo instancia [7].



En sistemas como Objetive-C [46] el servicio se proporciona a través del NSProx. Todos los objetos pasan por este mediador y de allí son enviados a las clases que operan en el programa. NsProx decide si un mensaje puede ser recibido o no. Si se da paso al mensaje, la subclase responderá con el método y el valor de retorno que tenga la invocación del objeto.

## 1.3 CARACTERISTICAS DE LOS LENGUAJES REFLEXIVOS

Los lenguajes que soportan reflexión en forma nativa deben soportar los conceptos estudiados anteriormente. Los cuales se caracterizan en:

- **Representación del sistema:** los lenguajes deben poseer mecanismos que permitan identificar los datos u operaciones en sus programas que van a ser de carácter reflexivo. Los sistemas implementados en estos lenguajes tienen la

posibilidad de desarrollar computación reflexiva incluyendo código que prescribe la manipulación de los datos.

- **Mecanismo de reflexión:** debe ser posible interrumpir la ejecución del programa e ir a la parte reflexiva donde el modelo computacional puede ser accedido y manipulado y luego retornar a la parte del programa. Esto puede ser hecho explícita o implícitamente. Un sistema utiliza reflexión explícita si la parte del programa redirecciona explícitamente el control a la parte reflexiva. Caso contrario, la reflexión se desarrolla implícitamente.

- **Conexión causal:** conjunto de instrucciones que permiten que el programa tenga una conexión con su parte reflexiva. De modo tal, que si se realizan cambios estructurales, todas las partes involucradas deben cambiar en consecuencia.



## 1.4. ARQUITECTURA REFLEXIVA ORIENTADA A OBJETOS

En el paradigma de orientación a objetos la reflexión computacional está relacionada con la representación de las propiedades y comportamiento de las clases y objetos como datos, y la posibilidad de modificarlo dinámicamente [5] [6].

Los programas que tengan habilidades reflexivas deben poseer mecanismos que permitan mantener la información del programa así como la comunicación de éste con su parte reflexiva o conexión causal, pudiéndose identificar que un sistema se componga de dos partes como mínimo, una donde está el programa con todas sus estructuras de datos y demás procedimientos en memoria, y la otra donde se ubican los componentes activos reflexivos. Es decir, bajo la perspectiva de una arquitectura reflexiva orientada a objetos [1][42], un sistema se considera formado por dos partes: la parte correspondiente a la aplicación, y la parte reflexiva, la cual es capaz de razonar y actuar sobre el sistema. Esto permite separar lo que es

reflexivo de lo que es invariante, definiendo de una manera más sencilla el proceso de reflexión en el ámbito de la OO [3][10].

Un metasistema es un sistema computacional que tiene como dominio otro sistema computacional, llamado sistema objeto. Por tanto, un metasistema es un sistema que razona y actúa acorde con otro sistema computacional. Un metasistema tiene una representación de su sistema objeto en sus datos [5].

Su programa especifica una metacomputación acerca del sistema objeto y por tanto es llamado un metaprograma. La metacomputación retorna nueva información acerca del sistema objeto y actúa acorde con el sistema objeto. Por ejemplo, un metasistema es un sistema que se asimila a otro sistema. Un metaprograma es un programa que se aproxima a otro programa. La metacomputación es la computación que se asimila a otra computación. Por tanto, también se usarán los términos sistema objeto, programa objeto y computación objeto en el contexto de los meta sistemas [4][6][11].



Existe un enlace de conexión causal entre los datos del metasistema y lo que éstos representan, denominado atributos y relaciones en el sistema objeto. Ellos construyen modificaciones a su sistema objeto de tal forma que el comportamiento del sistema objeto se ve afectado.

El enlace de conexión por causalidad se establece por modificación del código del sistema objeto acorde con la forma como trabaja la metacomputación (figura 3). El sistema objeto puede cambiar solamente por modificaciones establecidas en un nivel meta.

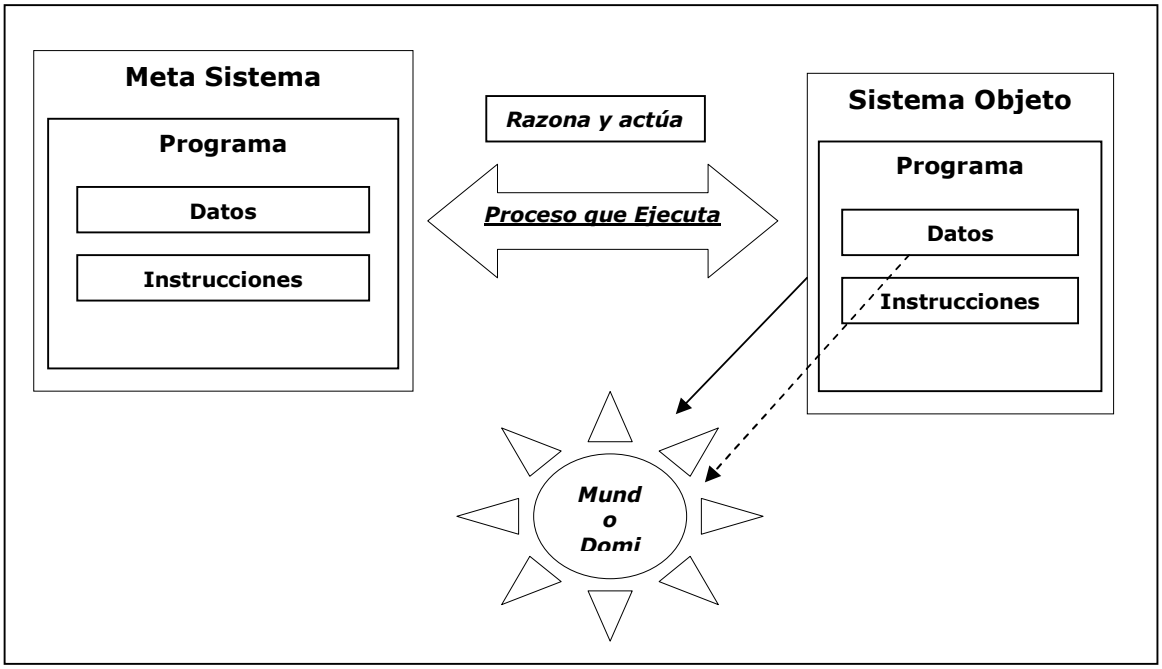



**Figura 3. Metasistema**



Un sistema operativo se ve como un metasistema, tiene un conjunto de sistemas objeto, llamados procesos. Tiene representaciones de esos procesos. Guarda información acerca de ellos tales como la fecha en que fueron creados, su estado actual, su prioridad, entre otros datos. El sistema operativo "razona" y "actúa" sobre estos procesos cuando decide cual activa o cual termina. Igualmente actúa sobre sus sistemas cuando inicia o termina su propia computación.

Estas partes residen en dos niveles diferenciados: el nivel base y el nivel meta, respectivamente. Los componentes relacionados con la funcionalidad de la aplicación se representan en el nivel base y son manipulados por el nivel meta. El nivel base no tiene conocimiento de la existencia del otro.

En el nivel meta se encuentran los objetos que son creados en el nivel base pero que pueden ser accedidos o no de manera reflexiva. Estos objetos son denominados meta objetos.

Los meta objetos pueden crearse directamente de un objeto del nivel base por lo que se llaman objetos reflejados (referente) o a partir de una clase que tiene correspondencia directa con una clase en el nivel meta llamada meta clase (véase figura 4).

Los objetos en el nivel meta pueden igualmente intercambiar mensajes y acceder a métodos dependiendo de los modificadores de acceso que se usen; es decir, las mismas propiedades que se encuentran en un objeto común (nivel base) se deben dar en el nivel meta por ejemplo con lo referente a la accesibilidad y visibilidad de los atributos y métodos [4].

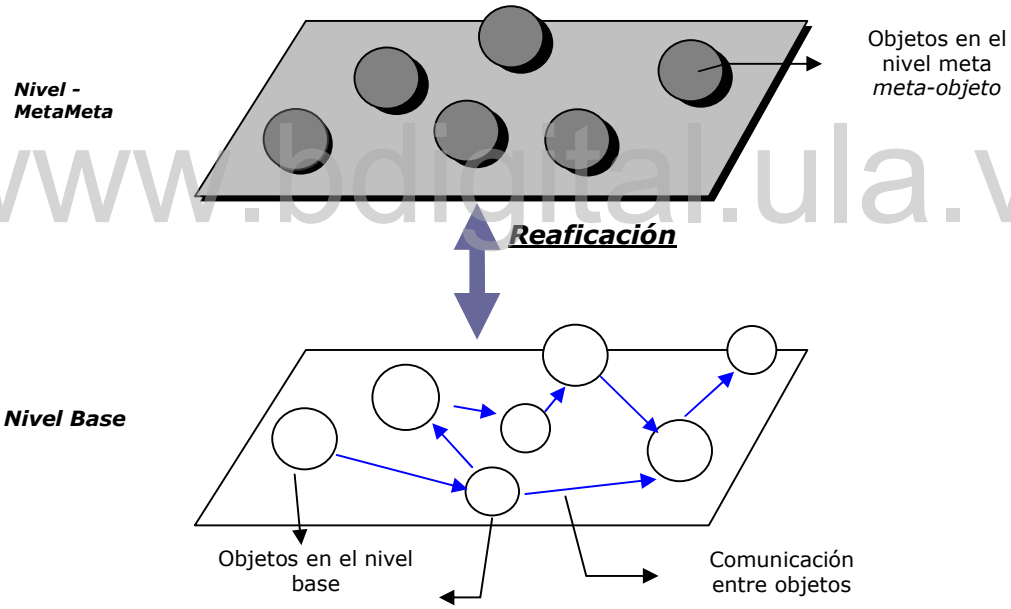





**Figura 4. Arquitectura reflexiva en OO (tomado de [2])**

Los niveles están conectados en una forma causal de manera tal que cambios en el nivel base son reflejados en el nivel meta. Esta conexión, como ya se mencionó, se denomina reificiación, y su fin es permitir reflejar los resultados al nivel base de los objetos que se encuentran en el nivel meta (metaobjetos).

El objetivo de la reificación es ofrecer semántica y procedimientos que permitan obtener datos de la computación abstracta que está en la parte reflexiva, según la arquitectura del lenguaje computacional que se use. Una tarea ideal sería permitir funciones de reificación a nivel estándar que permitan obtener entre otras cosas, estructuras de datos a partir de cualquier sistema abstracto o un procedimiento para generar código de cualquier programa.

En la actualidad muy pocos lenguajes orientados a objetos soportan reflexión computacional. Usando el modelo de dos capas e implementando el mecanismo de reificación se pueden otorgar características reflexivas permitiendo aspectos relevantes como:

- Capacidad para representar objetos y clases como un dato abstracto.
- Intercambio de mensajes entre objetos del nivel base y objetos de nivel meta.
- Distinguir el flujo de ejecución de un programa en parte reflexivo y su ejecución normal.



Los niveles estarán estructurados como una red de objetos que se comunican, los cuales pueden intercambiar y acceder a la información de la clase, del objeto o del método. Explicación que se da a continuación:

### 1.4.1 Reificación a nivel de clase (metaclase)[6][7].

Los objetos que se encuentran en el nivel meta son creados a través de una clase denominada metaclase que corresponde a una clase del nivel base. La metaclase puede crear de la misma manera como se instancia un objeto en un entorno no reflexivo.

El nivel base manipula la metaclase que se asocia a la clase reflejada. Se instancia la metaclase creando el meta-objeto y se opera con los servicios del metaobjeto como si fuera un objeto de la clase reflejada (figura 5). De esta manera, es posible

incorporar nuevo comportamiento a un sistema sin necesidad de realizar modificaciones en las clases de la aplicación. Los pasos para incorporar nuevo comportamiento son: crear el metaobjeto que incorpore la nueva funcionalidad y establecer la asociación entre el nivel base y meta.

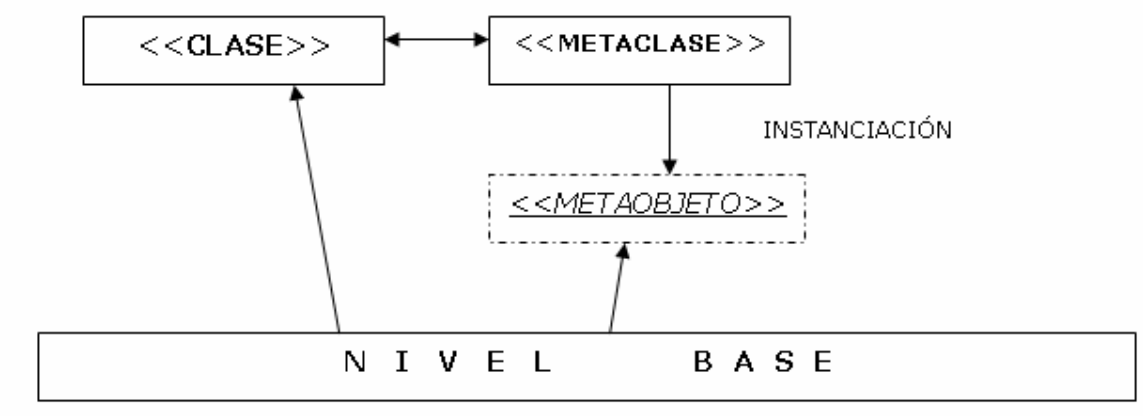



**Figura 5. Modelo de metaclase.**



Las meta clases tienen información sobre los aspectos estructurales de los objetos del nivel base. Si esta información es modificada, la estructura de los objetos es modificada en consecuencia. Este modelo permite a los diseñadores extender la parte estática de los lenguajes orientados a objetos [5].

### 1.4.2 Reificación a nivel de objeto (metaobjeto).

Se realiza una comunicación directa entre los objetos del nivel base con objetos del nivel Meta, es decir, existe un objeto perteneciente al dominio local y existe otro objeto que es reflejado a partir de éste al nivel Meta. Se establece una relación de "pares"[8] permitiendo que el comportamiento del objeto se determine por medio del metaobjeto que lo manipula y define las operaciones del objeto [7].

[8] El término *pares* hace referencia a que existe una clase base que se comunica con una clase del nivel meta en una relación de "clientela" permitiendo intercambio de mensajes bidireccionalmente.

Un metaobjeto puede hacer referencia a múltiples objetos. Cada objeto puede estar conectado (causalmente) a distintos metaobjetos. Se distinguen básicamente dos objetos: uno perteneciente al de nivel base y otro que es reflejado o también llamado referente como se muestra en la figura 6.

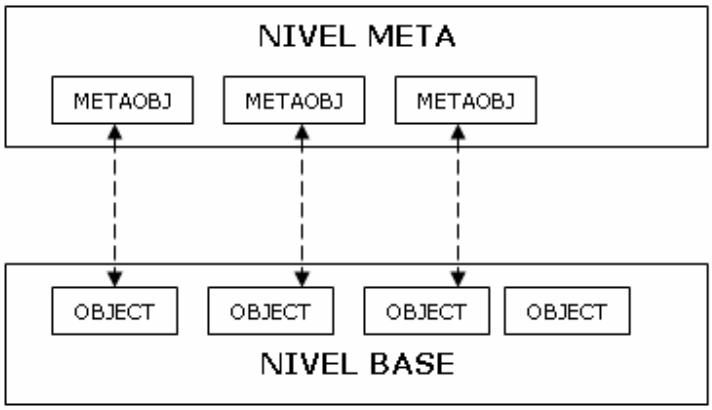



**Figura 6. Modelo de Meta Objeto**



## 1.5 COMPORTAMIENTO DE LA REFLEXION EN OO [1][2][4]

En la OO la reflexión se puede dar a nivel de: la clase, el método y del objeto, por lo que resulta necesario identificar el comportamiento e instante en que el control es redireccionado al nivel meta. A continuación cada comportamiento es presentado:

- **Comportamiento a nivel de la clase:** en reflexión de clase todos los objetos de una clase, están involucrados en el comportamiento reflexivo. Cada vez que un objeto de una clase reflejada recibe un mensaje, el mecanismo de reflexión redirecciona el control al mismo metaobjeto en el nivel meta. En reflexión de clase la asociación entre el nivel meta y base es establecida entre la clase reflejada y el metaobjeto que incorpora comportamiento a la clase.

- **Comportamiento a nivel de método:** en la reflexión de método solamente un método, denominado método reflejado o reflexivo, es afectado por la reflexión. Esto significa que cada vez que un objeto reciba un mensaje

correspondiente al método reflejado, el mecanismo de reflexión es activado. Si un objeto de la misma clase recibe un mensaje cuyo método no ha sido reflejado, el mecanismo de reflexión no es activado. Otros métodos de la misma clase pueden ser reflejados y asociados con el mismo o diferente meta-objeto. De esta manera, en la reflexión de método, la asociación entre el nivel base y meta es establecida entre el método reflejado y un metaobjeto.

- **Comportamiento a nivel de objeto:** un objeto reflejado o reflexivo requiere comportamiento reflexivo. Cada vez que un objeto reflejado recibe un mensaje, el mecanismo de reflexión es activado redireccionando el control al nivel meta. Por otra parte, si un objeto no reflejado recibe un mensaje, el mecanismo de reflexión no es activado. Otros objetos de la misma clase pueden estar reflejados y asociados con el mismo o diferente metaobjeto. De esta manera, la asociación entre el nivel base y el meta es establecida entre el objeto reflejado y un metaobjeto.



## 1.6. TIPOS DE REFLEXION

La literatura identifica dos modelos reflexivos dentro de un paradigma orientado a objetos él modelo estructural y de comportamiento [10] [7].

- **Reflexión estructural:** se define como la capacidad de un lenguaje para proporcionar una reificación completa, es decir, del programa que se esté ejecutando, así como reificación de sus tipos de datos abstractos. La reflexión estructural siempre realiza manipulación sobre el código del sistema, permitiendo incluso que se añadan fragmentos de código para integrarlos en tiempo de ejecución; puede ser visto en lenguajes funcionales como Lisp o Lenguajes lógicos como Prolog donde las instrucciones manipulan la representación del programa.

  En lenguajes OO este tipo de reflexión resulta compleja, ya que muchos lenguajes compilados no tienen forma de representar y abstraer su código en

tiempo de ejecución; a excepción de lenguajes como SmallTalk, que posee mecanismos de reflexión estructural de forma nativa. Existen aproximaciones en lenguajes OO que permiten introducir instrucciones para identificar que parte del programa y sus datos serán puestas en los niveles meta; sin embargo, la representación del programa original es reescrito para ofrecer reaificación en el código del programa (ejemplo actual de *OpenC++*).

- **Reflexión de comportamiento:** se define como la capacidad de un lenguaje para proporcionar reificación total en la semántica e implementación, así como de sus datos en tiempo de ejecución. Este tipo de reflexión manipula el comportamiento del programa en su totalidad y todas las partes del programa se pueden manipular de forma reflexiva.

La diferencia entre uno y otro es la que se establece a nivel meta: si hay metaclases será el primer caso, si hay metaobjetos será el segundo. En un lenguaje reflexivo basado en metaobjetos, el comportamiento de cada objeto es determinado por el metaobjeto asociado que controla y define sus operaciones. De esta manera, es posible, por ejemplo, introducir nuevo comportamiento, cambiar el mecanismo de herencia, controlar la activación de los objetos o modificar la ejecución de los métodos [5].



Un protocolo de metaobjetos [7] brinda a los usuarios la habilidad de modificar incrementalmente el comportamiento e implementación de un lenguaje de programación y la habilidad de escribir programas con dicho lenguaje. El primer mecanismo está relacionado con la información que puede ser manipulada en el nivel meta.

El comportamiento computacional del nivel base es transformado en dato (reificación). Las partes reificadas constituyen la metainformación que hace posible desarrollar el comportamiento reflexivo.

Los otros dos mecanismos están relacionados con definir cómo la asociación entre ambos niveles, es implementada y el mecanismo de reflexión es activado.

# 2. VISION DE LA REFLEXION EN C++

C es un lenguaje de propósito general orientado a ser de multiparadigmas, soporta variedades de tipos, como programación estructurada, procedimental, programación genérica y en casos lejanos algo de programación funcional. Su principal uso se da en la OO con la creación de C++ como lenguaje para las implementaciones que se hacían a bajo de nivel, como sistemas operativos. C++ introduce conceptos como "genericidad", "plantillas", "funciones virtuales", sobrecarga de funciones y/u operadores y mecanismos de introspección simples ofrecidos con RTTI [12].

## 2.1 GENERALIDADES

En la actualidad lenguajes como Java o la versión de C en Microsoft C# han introducido el concepto de reflexión computacional a sus plataformas en forma nativa. Poseen toda una estructura que permiten representar la metainformación de las clases en su sistema; es decir, presentan cualquier tipo de reificación que hacen que métodos, objetos y atributos puedan ser vistos como datos. Estos lenguajes ofrecen este servicio debido a que su arquitectura sugiere el uso de una maquina virtual que soporta toda la ejecución e interacción de los objetos permitiendo de una manera no tan compleja establecer la conexión causal entre sus elementos [51].



¿Qué sucede en C++?, el lenguaje no cuenta con un soporte nativo de los mecanismos de reflexión , aunque, se pueden ver operaciones de introspección y reificación muy elementales, por ejemplo la función sizeof (..) para determinar el tamaño de un dato elemental y en ciertas operaciones que se pueden usar con RTTI. El lenguaje no tiene implementado mecanismos de reflexión, por lo que algunas soluciones hacen uso de modelos que permiten extender la semántica del lenguaje como el *OpenC++* [6][13].

Según lo ya mencionado, los sistemas reflexivos están compuestos por el nivel meta y base, donde uno contiene los objetos que fueron instanciados y creados en un proceso de reflexión y el otro nivel donde se interactúa con estos objeto. En el lenguaje Java, sus clases son envueltas por una super clase llamada *object* [49], que almacena la metainformación permitiendo reflexión estructural. Por ejemplo, dada una clase de nombre circulo (figura 7):

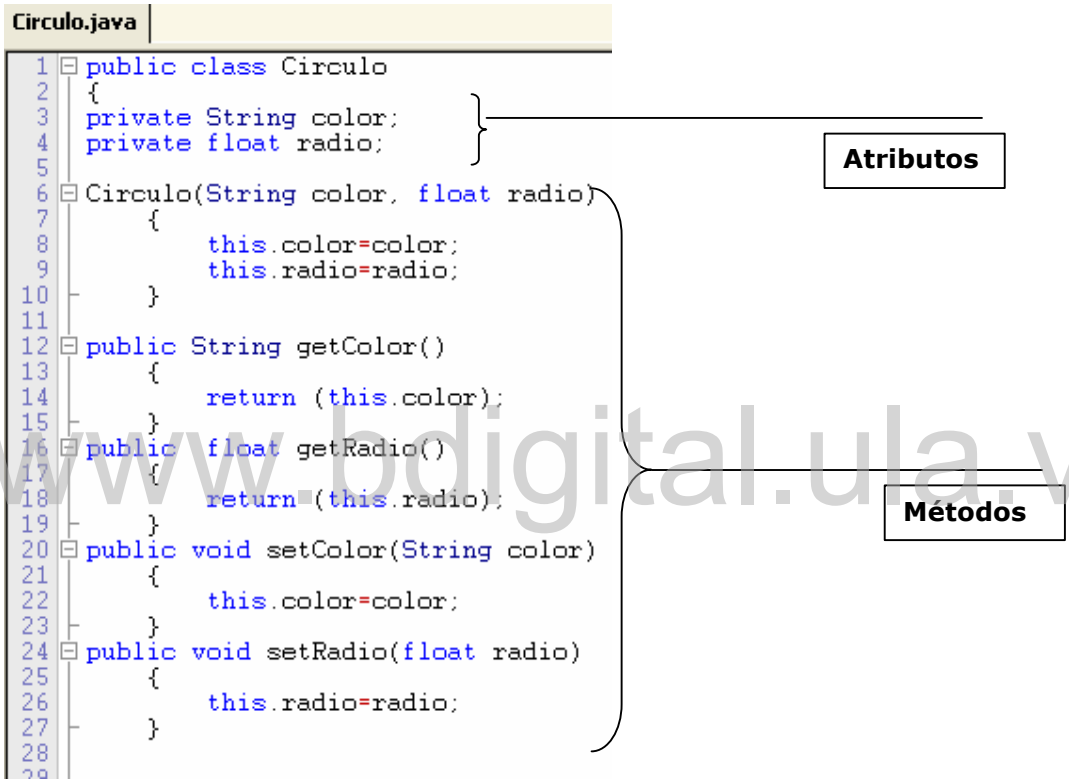



**Figura 7. Una Clase en Java.**

Se puede acceder a su información a través de la API de reflexión de java (java.lang.reflection) [52] que permite modelar la información de la clase, del objeto y de los atributos, ver figura 8.

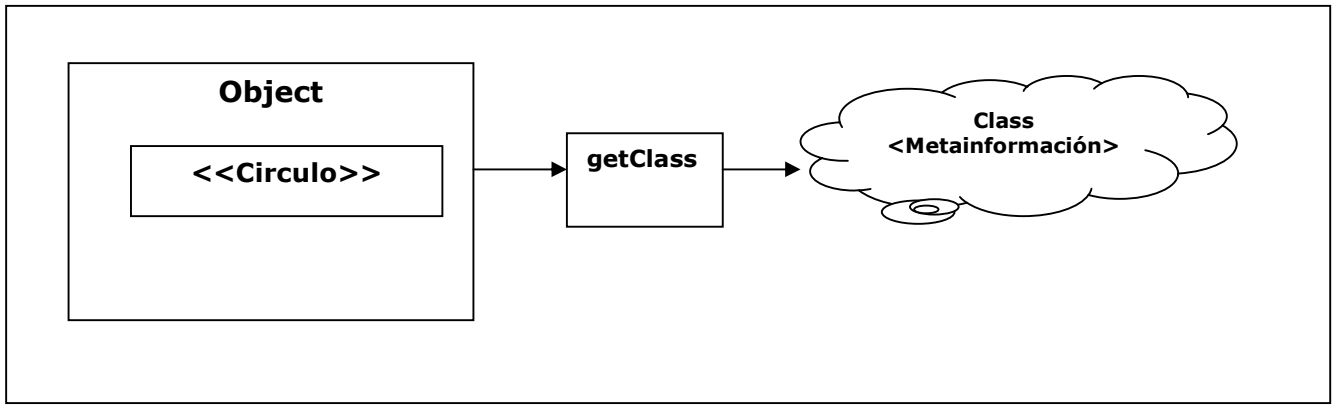



**Figura 8. Clase "Class" en Java**

Se cuenta con las clases *class*, *fields* y *methods* [50] que permiten recorrer de forma recursiva la estructura del objeto y realizar introspección y realización de la clase base. El metaobjeto es creado y a partir de *class* y su conjunto de clases se pueden invocar todos los servicios de la clase reflejada. Se toma como ejemplo la clase presentada en la figura 7 y se presenta el mecanismo de reflexión dado en Java (figura 9) y con esto lograr una visión de lo que se desea en la tesis.



PruebaReflexion.java

```
1  import java.lang.reflect.*;
2  public class PruebaReflexion
3  {
4
5
6      public static void main(String args[])
7          {
8              Circulo miCirculo=new Circulo("rojo", 3.4F);
9              imprimirObjeto(miCirculo);
10         }
11
12     public static void imprimirObjeto(Object objeto)
13         {
14
15         Class objetoDesconocido=objeto.getClass();
16         System.out.println("Nombre de la Clase: "+objetoDesconocido.getCanonicalName())
17         Field[] misAtributos=objetoDesconocido.getDeclaredFields();
18         Method[] misMétodos=objetoDesconocido.getMethods();
19         for (int i=0; i<misAtributos.length;i++)
20             System.out.println("Atributo->"+misAtributos[i].toString());
21         for (int i=0; i<misMétodos.length;i++)
22             System.out.println("METODO:->"+misMétodos[i].toString()
23         }
24 }
```

Clase de Nivel Meta

Clase de Nivel Base

Introspección

**Figura 9. Ejemplo del uso del paquete java.lang.reflect de Java**

En el ejemplo anterior se interactúa con una clase que representa un circulo (atributos color y radio, tipo string y float, respectivamente) y se invoca el método imprimir objeto. Note como por la propiedad de herencia la clase circulo es tomada por la clase *object* que representa cualquier clase en el mundo de Java; en las líneas "15" a la "18" de la figura 9, se crean objetos de las clases *Class, Field* y *Method*, respectivamente. Clases que permiten almacenar la información reflejada del objeto circulo. De esta forma, cada una de las partes de la clase resulta ser una clase independiente, pudiendo ser manipulada por el programa principal.

En la actualidad C++ no posee esta característica; sin embargo, existen soluciones que sugieren usar una extensión de C++ llamada *OpenC++* [14] que basa sus operaciones de representación de clases a través de protocolo de metaobjetos (MOP) [5]. *OpenC++* cuenta con una etapa de precompilación donde se obtiene y se almacena la metainformación del objeto, después se traduce el código *OpenC++* hacia C++ común para ser compilado de manera normal. La etapa de precompilación añade sentencias al código fuente de la clase, lo que produce que al terminar la compilación, el tamaño del código objeto aumente de manera considerable.



Tomando como referencia el ejemplo citado en la figura 9, se puede hacer una abstracción de los pasos que implican realizar reflexión en un lenguaje OO, estos podrían ser:

1. Pedir el tipo de objeto en el nivel base.
2. Obtener el descriptor de la clase, que proporciona información sobre los métodos de clase y sus atributos.
3. Con los descriptotes de la clase ir a buscar en el repositorio de la Metainformación, los campos del objeto y métodos
4. Instanciar clases y métodos.

El paso (1) involucra un proceso de reificación donde se hace necesario que exista un objeto que es reificado desde una clase del nivel base. La instanciación de la

clase constituye la reificación de sus atributos; es decir del objeto reflexivo. El objeto reificado tiene una conexión causal con sus atributos así como sus funciones miembros.

Se puede concluir que el programa escrito en C++ debe contar con mecanismos que permitan acceder a la información que solo es conocida por el compilador (caso de Java con su máquina virtual y su *bytecode*), en este caso se obtendría capacidad reflexiva a nivel estructural adecuada a los requerimientos de representar lo *no funcional* de un programa; es decir, la adaptabilidad que proporciona una aplicación reflexiva. Los mecanismos que C++ debería tener serían:

- Operación de introspección: el acceso a la información que aparece en las el código como identificadores de funciones y clases. Esta tarea se ve en la adquisición de la información de los datos en tiempo de ejecución y del tipo de información relacionada con la identificación de atributos y nombres de la clase que operan en el programa base y el alcance que estas tienen. Esta información incluye toda la especificación de qué hace el compilador sobre la clase y funciones; como por ejemplo: tipos de datos de los atributos, nombres de los métodos, identificadores de acceso de ámbito, tipo de dato de los parámetros, entre otros aspectos.

- Realización: capacidad para invocar funciones e instanciar de objetos adquiridos por introspección.

- Extensión: capacidad para influir en el comportamiento de una clase (en oposición a una instancia de una clase), alterando de manera eficaz la información que en el futuro se haría por introspección, sin compilar código adicional.

## 2.2 MECANISMOS DE INTROSPECCIÓN EN C++

De manera nativa en C y particularmente en C++, existen dos tipos de mecanismos de introspección [8][9]. Los demás mecanismos que existen son implementaciones y adaptaciones realizadas por los programadores. Estos mecanismos comprenden:

### 2.2.1 . Genéricas (Genericidad)

C++ ofrece un mecanismo denominado *templates*. Los *templates* son simples operaciones que definen el parámetro de un tipo de dato y cuáles métodos deben ser reemplazados por este tipo. También es conocido como parametrización, en términos generales es pasar como argumento un tipo de dato sea simple o una clase. Se usa el caracter "T" o "?" en la definición del *template* para indicar que no se conoce el tipo de dato [15][37]. El siguiente código ilustra el primer tipo de introspección por identificación de tipo:



```
template <typename T>  struct type_descriptor
{
    template <typename OS>
    OS& write(OS& os) const { return os << "unknown-type"; }
};

template <> struct type_descriptor<int>
{
    template <typename OS>
    OS& write(OS& os) const { return os << "int"; }
};

template <> struct type_descriptor<char>
{
    template <typename OS>
    OS& write(OS& os) const { return os << "char"; }
};

template <> struct type_descriptor<long>
{
    template <typename OS>
    OS& write(OS& os) const { return os << "long"; }
};
```

En este ejemplo, se contempla la identificación de un tipo de dato que se le pasa en la invocación del *template*, si no se reconoce la base del *template* retorna la cadena "*unknown-type*". Después, se pueden apreciar varias especializaciones que han sido definidas, estas son usadas para identificar el tipo exacto del parámetro y varias especializaciones que resultan recursivas para cada uno de los tipos de datos primitivos que restan.

Con este tipo de implementación se puede ir pensando en la solución de implantación de la reflexividad; cabe notar, que solo se esta haciendo ejemplificación de introspección por tipos y no de una reflexión total como amerita este trabajo.

### 2.2.2 RTTI



Denominado información de tipo en tiempo de ejecución o en inglés de donde provienen sus siglas *run time type information* incorporada en el lenguaje en marzo de 1993 [13].

Este sistema es una parte importante del mecanismo de comprobación de tipos del lenguaje y permite la identificación incluso cuando el objeto solo es accesible mediante un puntero o referencia. Por ejemplo, el sistema hace posible convertir un puntero a clase-base virtual en puntero a clase derivada.

Permite comprobar si un objeto es de un tipo particular y cuando dos objetos son del mismo o distinto tipo. Esto puede hacerse con el operador *typeid* [52]. El mecanismo RTTI forma parte de un sistema más amplio de funciones y/o clases de la librería estándar del C++ que proporcionan determinadas funcionalidades en tiempo de ejecución.

La utilización del mecanismo RTTI produce cierta sobrecarga en tiempo de ejecución, por lo que la mayoría de los compiladores disponen de una opción para habilitarlo o deshabilitarlo a voluntad [13].

El siguiente ejemplo muestra identificación de tipos usando *typeid*:

```
#include <iostream>
#include <typeinfo>
class A {
public:
      virtual ~A() {}
};
class B : public A {
};

void printType(const A& a) {
      using namespace std;

      if (typeid(a) == typeid(B))
            cout << "Objeto de tipo B\n";
      if (typeid(a) == typeid(A))
            cout << "Objeto de tipo A\n";
}

int main() {
      B b;
      printType(b);
      return 0;
}
```



El ejemplo anterior se muestra una consulta a través de la función *typeid* donde se obtiene el tipo de objetos en tiempo de ejecución. Permite comprobar si un objeto es de un tipo particular, y si dos objetos son del mismo tipo.

RTTI ofrece un mecanismo denominado "moldeado dinámico" que consiste en intentar hacer una conversión a un tipo de datos en concreto, el valor de retorno será un puntero al tipo de dato deseado, sólo si el tipo es adecuado y tiene éxito; de otra forma devuelve cero para indicar que no es del tipo correcto.

El operador *dynamic_cast* [52] permite conversiones de punteros y referencias a clases, aunque exige que los punteros y referencias se refieran a clases de la

misma jerarquía. De no cumplirse está condición, la conversión es imposible y según los casos, el operador produce un resultado nulo o lanza una excepción. Cuando se trata de clases pertenecientes a una jerarquía, hay que distinguir los casos en que las conversiones (*cast*) entre punteros o referencias se realizan en sentido descendente (*down*) desde las clases-base hacia las derivadas, o ascendente (*up*), desde las clases-derivada hacia las superclases. Las palabras ascendente y descendente son usadas debido a que las jerarquías de clases se representan como árboles invertidos con la clase-base en la parte superior, y las derivadas hacia abajo. Las conversiones en sentido descendente se denominan *downcast*, y las contrarias *upcast*. Finalmente, cuando la conversión se da entre clases hermanas, se denomina *crosscast*. El siguiente ejemplo muestra la utilización del operador *dynamic_cast*:

```
#include <iostream>
using namespace std;

class Circulo { public: virtual ~Circulo(){}};
class FiguraCircular1 : public Circulo {};
class FiguraCircular2 : public Circulo {};

int main() {
  Circulo * b = new FiguraCicular1; // Upcast
  FiguraCircular1 * d1 = dynamic_cast<  FiguraCircular1*>(b);
  FiguarCircular2 * d2 = dynamic_cast<  FiguraCircular2*>(b);
}
```



El `main()` tiene un puntero a `FiguraCircular1` que es elevado a `Circulo`, y después se hace un *downcast* tanto a puntero `FiguraCircular1` como a puntero a `FiguraCircular2`.

Como se pudo notar, C++ solo ofrece introspección a nivel de identificación de tipos, el lenguaje no provee introspección estructural ni remotamente introspección por comportamiento y es claro que las aplicaciones requieren más que simples consultas de tipo o *casting* entre objetos.

## 2.3 TIEMPO EN QUE PUEDE OCURRIR LA REFLEXION

En el capitulo anterior se analizaron los diferentes tipos de reflexión que se pueden encontrar en los sistemas; sin embargo, se puede hacer un eje transversal y dividirlas en dos grandes grupos: la reflexión que se ofrece en tiempo de compilación y la que se da en tiempo de ejecución.

### 2.3.1 Reflexión en tiempo de compilación

La reflexión en tiempo de compilación generalmente ofrece mecanismos para realizar introspección estructural sobre el código fuente. Los mecanismos ofrecen información que es derivada directamente de los tipos de datos, variables y estructuras de control [12].



Esta información es usada para alterar el código en el momento que se desee, por ejemplo, para instanciar plantillas o generar datos externos que se necesiten en un proceso de precompilación o cualquier representación que genere la metainformación que permita consultar o alterar el programa.

La información que se necesita está disponible en las macros del preprocesador o localizaciones (direcciones) físicas; por ejemplo, el nombre del código fuente o el número de líneas que se tienen, igualmente con las instrucciones que requieren uso de *templates* donde las instancaciones dependen en la gran mayoría de los casos de la sobrecarga de operadores y de funciones.

La idea sugiere netamente una etapa de precompilación para realizar el almacenamiento de la metainformación a través de prepocesadores externos, con esto, la información que no es disponible al programador se puede obtener de manera menos compleja. Si este problema es resuelto, el mecanismo de reflexión quedaría funcionando a nivel estructural.

### 2.3.2 Reflexión en tiempo de ejecución

La reflexión en tiempo de ejecución provee mecanismos cuando el programa se está ejecutando [2]. La información generalmente incluye objetos, especificación de clases o solo la obtención de un valor de una variable en particular; adicional se podría también pensar en obtener información directamente relacionada con código de depuración (*debugging*) o del contenido de los registros del procesador.

Sus servicios comprenderían la alteración de valores, llamado e instanciaciones de funciones, métodos, clases y la modificación de lo que se tiene disponible en ejecución. Con respecto a la información de *debugging* la reflexión proporcionaría operaciones para auditar los eventos de una clase, como borrado, instanciación e intercambio de mensajes entre objetos.

## 3. SOLUCIONES PARA PROPOCIONAR REFLEXION EN C++

Las soluciones en C++ son variadas. Se presentan a continuación las más relevantes. La primera solución y ya tratada es la dada por RTTI, que consiste simplemente en introspección a nivel de identificación de tipos.

Cada una de las soluciones propone su forma particular de mantener la información que solo puede ser manejada por los compiladores, las diferentes opciones e implementaciones que se ofrecen a nivel de obtención de datos son:

1. Información de depuración: ofrece las ventajas que el sistema no es cambiado y es posible extraer información completa acerca de los tipos usados en el programa. Desventaja: Los programas al compilarse deben habilitar la opción de depuración y analizar que tipo de código es generado; además el sistema base debe conocer el tipo de información que manejan los registros de depuración, generalmente complejas de interpretar [38].

2. Preprocesadores: los programas permanecen sin alterar su semántica. La desventaja de esta implementación es que se necesita de un *parser* para identificar cada una de las partes del programa [39].

3. Compiladores especializados que soporten reflexión. Los usuarios solo escribirían sus códigos y el sistema se encarga de generar código para los métodos reflexivos. La desventaja de este enfoque, es que realizar un compilador para tal tarea amerita un tiempo considerable de desarrollo y actualmente ya se cuentan con compiladores que tienen esa función incorporada como el caso de Visual C++.

4. Librerías para la construcción de la meta información. Esta opción es las mas viable, ya que, los programadores serían los que desarrollarán

algoritmos que permitiesen simular un sistema reflexivo. Cada programador realizaría su librería en el nivel de reflexión que necesite para su aplicación.

## 3.1 MECANISMOS DE TRANSFORMACIÓN DE CÓDIGO

Trabajan generalmente en tiempo de compilación y usan preprocesadores de código fuente para proveer una interfaz de introspección y otra de modificación, usando el protocolo de meta objetos (MOP). La implementación ofrecida en C++ es *OpenC++* [36] que realiza el enlace entre el nivel base y el meta en tiempo de compilación, el funcionamiento se describe en la figura 10.

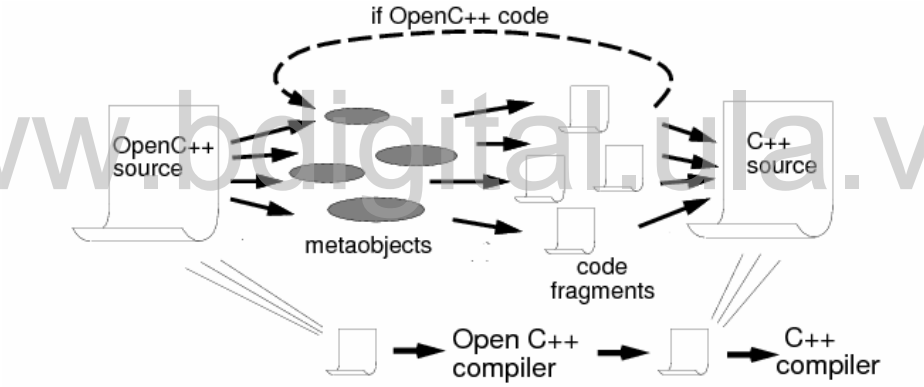





**Figura 10. Esquema de manejo de Open C++** (tomado de [36])

Para operar con *OpenC++*, el código debe definir los objetos reflejados y colocar la información respectiva para que sea interpretada por el preprocesador. La estructura del programa se coloca en un grafo y es el programador quien debe almacenar la información.

El compilador de *openC++* generará código C++ que será a su vez recompilado. En la primera traducción, existen una serie de primitivas en tiempo de compilación que son llamadas al crearse el árbol sintáctico. Se puede generar el código C++ que se desee modificando así la semántica del código fuente. El código generado puede

apoyarse en otras librerías adicionales como por ejemplo la implementación de un sistema de persistencia. El código generado es muy eficiente en comparación con el resto, pero es mucho menos flexible puesto que necesita recompilar el código fuente cada vez que se desee modificar la semántica de un programa.

## 3.2 LIBRERIAS DISPONIBLES

Librerías que permiten simular los mecanismos de introspección y realización. Las soluciones se basan en el concepto de metaprogramación o programas que escriben programas usando *templates*. El código de la clase es reescrito para ser reconocido por la librería.

### 3.2.1.METACLASES Y REFLEXION DE VOLLMAN



Se define una API con la que se construye tanto el sistema base como el meta sistema. La API ofrece dos clases principales *ClassDef* y *DataMember* que representan la clase, atributos y métodos. Se debe reescribir el código de las clases que se desean reflejar.

Ventajas:

- Sistema reflexivo en tiempo de ejecución.
- Mantenimiento de la conectividad causal.

Desventajas:

- El programador debe usar la API para construir su aplicación. Vollmann [40] propone soluciones para poder integrar el sistema con aplicaciones desarrolladas de la forma tradicional.
- Sobrecarga en memoria y en tiempo de ejecución.
- El código resultante es complejo ya que no existe una visión directa de los tipos de datos usados.

### 3.2.2. *REFLEX C*++[41]

Propuesta desarrollada por la Organización Europea para la Investigación Nuclear (CERN). Soporta todas las posibles estructuras del lenguaje. Necesidad de un preproceso mediante una herramienta que lee el código y genera la metainformación de la clase en un archivo XML. Posee las siguientes características:

- Maneja las normas propuestas por la ISO/IOC para C++
- Genera información de forma automática para poder tener operaciones de reflexión en el nivel base.
- Se basa en el concepto de diccionario de datos usando XML; es decir, la clase se transforma en un archivo XML que describe sus partes, el proceso se realiza con la herramienta g*cc-xml* [47].
- Optimiza el uso de memoria.
- Las operaciones de introspección las realiza directamente en los diccionarios que genera.

## 4. DISEÑO DE LA SOLUCION

Ser capaz de acceder a la información de una clase, así como de manipularla es el objetivo principal de las operaciones en una tarea de reflexión. Un mecanismo de reflexión básico debe permitir: consultar una clase y producir su invocación; Se podría pensar en un esquema general de arquitectura reflexiva para C++, véase figura 11. Se tiene una clase persona y un objeto de este tipo objPersona con atributos nombre y cédula y sus métodos de acceso y modificación de atributos[9].

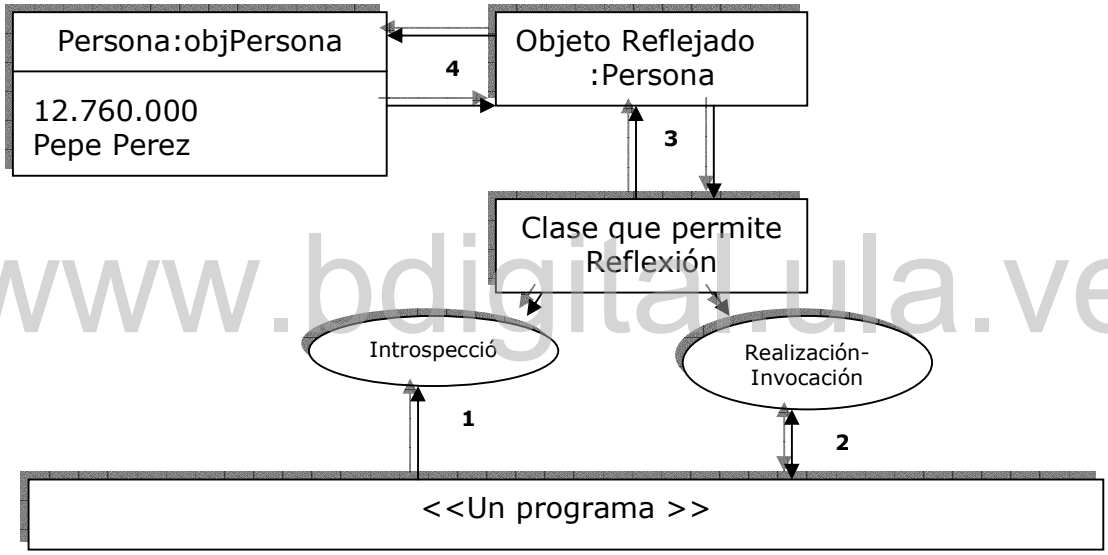





**Figura 11. Un ejemplo de Solución Reflexiva en C++**

De la figura anterior, se tiene un programa en el nivel base que desea conocer cuáles métodos y atributos tiene una clase de nombre persona (introspección, Paso 1), el programa puede instanciar la clase con el mecanismo de realización ofrecido en el paso 2. El objeto reflejado, paso 3, se comunica con el programa a través de los servicios ofrecidos por la clase que permite la reflexión. El objeto reflejado (nivel meta) tiene una referencia a un objeto tipo persona con el cual puede acceder a sus

[9] Métodos que son identificados con los caracteres get…( ) y set…( ) por cada atributo que se desee acceder y modificar, respectivamente.

datos y modificarlos (paso 5). A nivel código el nivel base podría usar instrucciones como:

*…objetoDeClaseQuePermiteReflexión.obtengaClase("Persona");*
*…objetoDeClaseQuePermiteReflexión.obtengaClase("Persona").obtengaMétodos();*
*…objetoDeClaseQuePermiteReflexión.obtengaClase("Persona").obtengaAtributos();*

El programa no tiene conocimiento de las operaciones de la clase[10], en una operación convencional, el programador invocaría los métodos de forma nativa: creando el objeto de tipo persona y llamando sus métodos, pero en el enfoque presentado en la figura 11, solo llamaría la clase que permite la reflexión y la clase reflejada podrá ser llamada de forma dinámica, sin necesidad de escribir código ni de recompilarlo. Con esta idea se puede comenzar con el desarrollo de la solución.

La implementación propuesta se denomino sistema de introspección y realización en C++ abreviada como SIRC. Se desarrolló usando metodologías ágiles donde el ciclo de vida comprende el análisis del problema, el diseño, la construcción y pruebas [53].



## 4.1 MECANISMO DE INTROSPECCION

El sistema reflexivo debe soportar el mecanismo de introspección o la capacidad para realizar consultas sobre una clase y sus partes, también la realización o invocación, que es la capacidad para invocar servicios y/u operaciones de forma dinámica.

Por efectos de complejidad en la implementación, se toma como referente a nivel de la introspección las operaciones de: identificación de tipos y de identificación estructural. Por parte de la realización, se toman invocaciones a métodos que no

[10] El termino "Conocimiento" no se hace ninguna alusión a programas con inteligencia artificial solo se ejemplifica el hecho de que el programador no conoce la clase ni los servicios que se desea instanciar.

desencadenen operaciones que tengan que ver con otros objetos, solo operaciones que estén definidas en el mismo ámbito de la clase.

### 4.1.1 Análisis del problema

La información sobre las propiedades de un objeto, sus atributos, su estructura, y demás características, se denomina metainformación. La metainformación es información que permite que el nivel base obtenga consultas sobre la definición del objeto[11] y permite conocer cuales objetos serán o no reflejados desde el nivel base al nivel meta.

En C++ esta información es capturada en la definición de la clase, por lo que las características de la metainformación se dan en el momento de su desarrollo.



Un problema a solventar está dado en el hecho de que la información no está disponible en tiempo de ejecución. Consecuentemente, es imposible manipular las clases como objetos y no es posible añadir nuevas clases. La idea básica de la reflexión es que el programa (nivel base) pueda ver u observar (introspección) las clases y métodos como si fueran datos (reificación); esto es, que las definiciones de clases de objetos y sus atributos se puedan manipular en tiempo de ejecución.

Si se desea analizar y comprender lo que contendrá la metainformación esperada se debe comenzar por interpretar qué partes de la clase son necesarias mantener. La figura 12 muestra una clase de nombre persona que se tomará para explicar el contenido que contendrá la metainformación.

[11] Se usa el término "definición del objeto" para referirse a la clase que permite crear e instanciar el objeto.

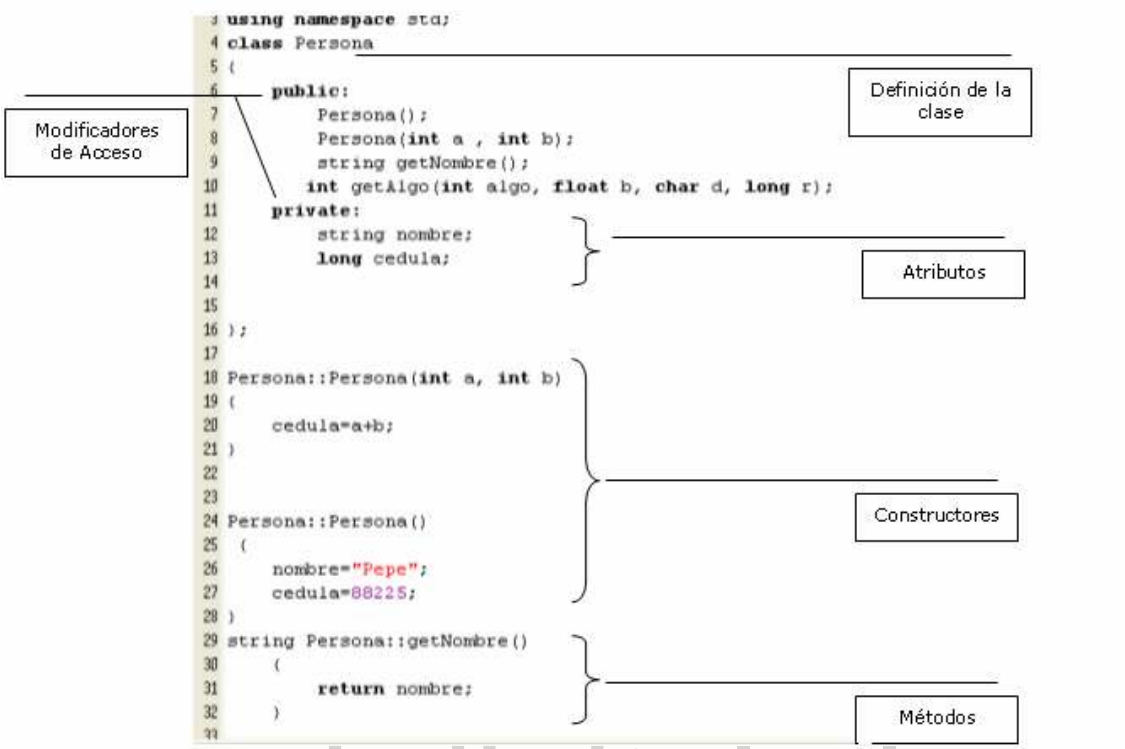





**Figura 12. Partes de una clase en C++**

Una clase es una plantilla que define la estructura de un conjunto de objetos, que al ser creados se llamarán las instancias de la clase. Esta estructura está compuesta por la definición de los atributos y la definición e implementación de las operaciones (métodos) [14]. Las clases poseen atributos (figura 13):

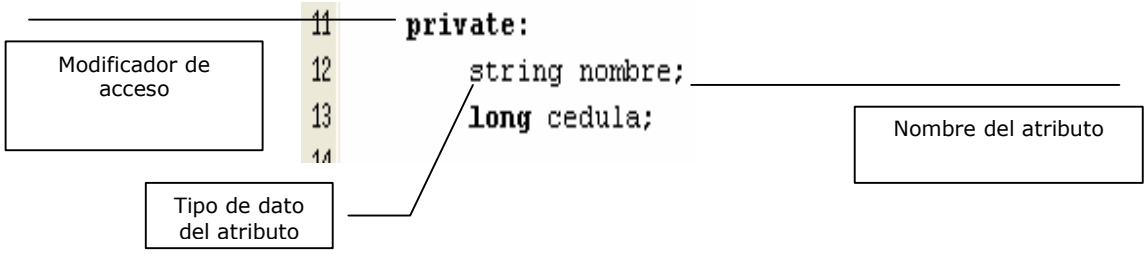



**Figura 13. Un atributo de una clase en C++**

Las clases tienen métodos de los cuales, algunos pueden ser constructores (método especial que sirve para construir la clase e inicializar sus atributos):

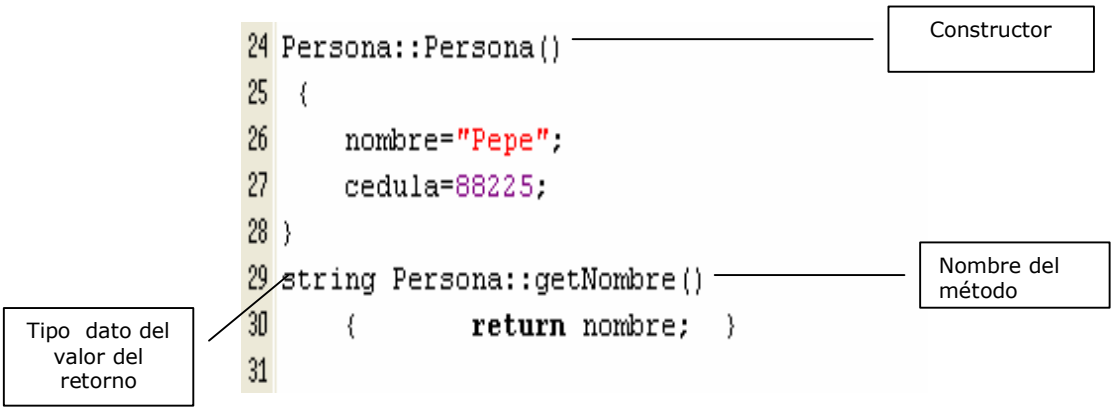



**Figura 14. Métodos de una clase en C++**

Tomando como referencia las figuras 12, 13 y 14 que muestran la información que contiene una clase, se puede obtener una visión de la información que se tiene a través de cualquier mecanismo introspectivo básico. Este comprenderá, por parte de la clase:



**Tabla1. Operación de Introspección sobre una clase**

| NOMBRE | ATRIBUTOS | FIRMA DEL MÉTODO | CLASE DERIVADA |
|---|---|---|---|
| Persona | • Nombre<br>• Cédula | • Persona()<br>• Persona(int, int)<br>• string getNombre()<br>• int getAlgo(int, float, char, long) | No es un clase derivada |

Para los métodos:

**Tabla 2. Operación de introspección sobre un método**

| NOMBRE | ACCESO | TIPO RETORNO | PARAMETROS |
|---|---|---|---|
| Persona | public | void | Sin parámetros |

| | | | |
|---|---|---|---|
| Persona | public | void | Int , int |
| getNombre | public | string | Sin parámetros |
| getAlgo | public | int | int,float, char, long |

Para los atributos:

**Tabla 3. Operación de introspección sobre atributos**

| NOMBRE | ACCESO | TIPO |
|---|---|---|
| nombre | private | String |
| cedula | private | Long |

Tal información sugiere que la implementación debe conceder todas las posibles consultas descritas en las tablas 1, 2 y 3 y encontrar la estrategia de solución para realizar la introspección.



#### 4.1.1.1 Estrategia de solución.

Se debía buscar la forma de explorar una clase en tiempo de ejecución o en tiempo de compilación. En el primer caso, en tiempo de ejecución, consistiría de manera muy general en analizar las direcciones físicas de los objetos. Como por ejemplo con la manipulación de ELF[12] que permite mantener la información de enlazado y con esto encontrar las direcciones de los objetos, sus métodos y atributos y si es posible llegar al proceso de realización (invocación). El inconveniente de esta solución es que se tendría que buscar otro mecanismo para realizar la introspección para leer directamente la información del objeto.

La solución debía fundamentarse en las operaciones principales de un sistema reflexivo sin llegar a ser una solución invasiva o de reestructuración de código como se estudio en el capítulo 3. Para la implementación de SIRC se toma la base

[12] Es un formato de archivo para ejecutables, código objeto, librerías compartidas y volcados de memoria.

proporcionada por el CERN en su implantación llamada *Reflex* [41], que usa el concepto de diccionario, donde la clase se convierte en un archivo XML y el programa lee los datos de la clase de dicho archivo.

#### 4.1.1.2 Generación del diccionario

Para la generación del diccionario se usó una representación en XML, que es un metalenguaje[13] extensible de etiquetas que busca dar solución al problema de expresar información estructurada de la manera más abstracta y reutilizable posible. El término información estructurada quiere decir que se compone de partes bien definidas, y que éstas partes se componen a su vez de otras partes, lo que conlleva a tener un árbol de trozos de información [17].

Para la generación de la persistencia de la clase en el diccionario se usó la herramienta GCC-XML [47]. GCC-XML es una herramienta que trabaja con el lenguaje C que permite convertir un archivo que contiene la descripción de las clases, métodos y atributos del código fuente en un archivo XML. El propósito es generar una descripción XML del programa en una representación interna. Se realiza en XML ya que resulta fácil de analizar y se pueden crear programas que permitan leer e interpretar el archivo. GCC-XML se encuentra disponible en la Web, su instalación y configuración se realiza de forma automática por cualquier gestor de paquetes o a través de sus fuentes. El único requisito es tener el compilador *gcc* o *g++* instalado y actualizado. Así pues, la clase se convierte en la representación estándar que posee GCC-XML. Tomando como ejemplo la clase de la figura 12, su representación en XML se presenta en la figura 15.



[13] un metalenguaje es un lenguaje usado para hacer referencia a otros lenguajes. Los modelos formales de sintaxis para la descripción de la gramática[18].


**Figura 15. Representación de una clase en XML**

Especificando los elementos principales:

- <*Method*>: Información del método.
- <*Field*>: Información del atributo.
- <*Constructor*>: Información de los métodos constructores.

Cada uno de los elementos contiene un atributo "id" que es el identificador de la localización de la información de las partes de la clase o del programa, conformado por el caracter "_" seguido de un entero auto incrementado y un atributo de nombre "name" que especifica el nombre de la parte de la clase (nombre de la clase, de métodos, de argumentos).

De la marca <*Class*>:

- <*members*>: identificadores de la localización de la información de las partes de la clase (atributos y métodos).
- <*bases*>: Identificador de la localización de la información de la clase padre(si existe).
- <*Access*>: Información del modificador de acceso o de ámbito de la clase.

De las marcas <*Method*>, <*Field*> y <*Constructor*>:

- <*Argument*>: Información de los argumentos.
- <*Access*>: Información del modificador de acceso o de ámbito de los métodos y atributos.

De la marca <*Argument*>:

- <*type*>: Identificador de la localización de la información del tipo de dato.



De la marca <*type*>: Representa todas las declaraciones de los tipos de datos primitivos, objetos, arreglos, apuntadores y *structs*. <*type*> contiene atributos que contienen la información del tipo del dato y su tamaño.

La figura 16 muestra el elemento <Class> y sus atributos:

Class

| | |
|---|---|
| id | _1 |
| name | Persona |
| context | _2 |
| mangled | 7Persona |
| location | f0:5 |
| file | f0 |
| line | 5 |
| artificial | 1 |
| size | 64 |
| align | 32 |
| members | _3 _4 _5 _6 _7 _8 _9 _10 |
| bases | |

**Figura 16. Elemento class**

El atributo *members* almacena los valores de los atributos "id" de cada elemento que representa para la clase, la información de sus atributos (elemento *field*), métodos (elemento *method*) y constructores (elemento *constructor*). La figura 17 muestra la información de los atributos de la clase.

**Field** (2)

| | = id | = name | = type | = offset | = context | = access |
|---|---|---|---|---|---|---|
| 1 | _3 | nombre | _12 | 0 | _1 | private |
| 2 | _4 | cedula | _13 | 32 | _1 | private |

**Figura 17. Elemento Field**

De la figura anterior el atributo *type* tiene valores de _12 y _13, respectivamente, que corresponden a la definición de un tipo de dato del atributo de la clase. Por ejemplo, si se desea conocer el tipo de dato del atributo nombre de la clase persona, se buscaría el elemento cuyo atributo "id" tenga el valor de _12 ó _13 (véase figura 18).



**Typedef**

| | |
|---|---|
| = id | _12 |
| = name | string |
| = type | _11 |
| = context | _18 |
| = location | f1:60 |
| = file | f1 |
| = line | 60 |

**FundamentalType**

| | |
|---|---|
| = id | _13 |
| = name | long int |
| = size | 32 |
| = align | 32 |

**Figura 18. Elementos Typedef y FundamentalType**

#### 4.1.1.3 Tratamiento del Diccionario XML

Después de convertir la clase en el XML generado por gcc-xml el paso siguiente es crear un *parser*[14] que permita tomar cada uno de los campos y colocarlos en el sistema reflexivo[15].

Para extraer la información que contiene un documento XML se podría escribir código para analizar el contenido del archivo, pues es un documento de texto plano. Sin embargo, esta solución no es muy aconsejable y desaprovecharía una de las ventajas de XML: el ser una forma estructurada de representar datos.

La mejor forma de recuperar información de archivos XML es utilizar un *parser* de XML, que sea compatible con el modelo de objeto de documento (DOM). DOM define un conjunto estándar de comandos que los parsers devuelven para facilitar el acceso al contenido de los documentos XML desde sus programas. Un analizador de XML compatible con DOM [19] toma los datos del XML y los expone mediante un conjunto de objetos que se pueden programar.



Para llevar a cabo la tarea del Parser xml se uso un toolkit de nombre Libxml2 desarrollado por el proyecto GNOME. Libxml2 permite leer y hacer búsquedas sobre un archivo XML[20]. Es una librería que implementa un analizador sintáctico XML que permite fácilmente hacer uso de XML en aplicaciones. Es de agradecer que, a pesar de haber sido desarrollada en el seno del proyecto GNOME, esta librería no tiene ninguna dependencia con este entorno, por lo que puede ser usada perfectamente en aplicaciones totalmente ajenas a GNOME, si así se desea. Libxml incluye todo lo que cabría esperar de una librería de estas características; es decir, con libxml se puede leer, validar y modificar documentos XML, todo ello de una forma bastante clara y sencilla, permitiendo:

- Parsear el documento.

[14] Un *parser* es un módulo, biblioteca o programa que se ocupa de transformar un archivo de texto en una representación interna

[15] El término "Sistema Reflexivo" hace referencia a la librería o componente que se irá a crear como objeto principal de estudio de la Tesis.

- Extraer un texto.
- Extraer el valor de un atributo.

Libxml posee dos tipos de datos básicos: xmlDocPtr, usado para representar un documento XML, y xmlNodePtr, que se usa para representar cada uno de los elementos del archivo. Los dos tipos de datos son simplemente, punteros a dos estructuras que están definidas en los ficheros de cabecera de libxml, y más concretamente en el fichero *tree.h*.

Por ejemplo, si se tiene un archivo de nombre "Persona.xml" y se desea mostrar la información de una persona usando libxml.

```
<Personas>
    <Persona nombre=”Pepe” cedula=”88225”/>
</Personas>
```



Cargar el fichero XML:

```
1. xmlDocPtr doc;
2. doc = xmlParseFile("Persona.xml");
   3. if (!doc)
          cerr<<"Error al cargar documento XML”;
```

La línea (1) declara un tipo de dato `xmlDocPtr` que representa el documento y en la línea(2) se carga la referencia al documento en la variable"doc". Si `xmlParseFile(...)` retorna NULL el archivo no existe. Una vez que se tiene el documento XML cargado en memoria, sólo faltaría acceder a su contenido. Para ello, se usan los campos de la estructura `xmlDoc` y `xmlNode`. El paso siguiente será obtener una referencia al nodo raíz del documento para recorrer y buscar la información en cada elemento del XML. El proceso en código sería:

```
4. xmlNodePtr root=xmlDocGetRootElement(doc);
```

Para ubicarse en el atributo de cada elemento, se usa el tipo de dato xmlAttr, que representa la dupla atributo/valor de la siguiente manera:

```
xmlNode *child;
for (child = root->children; child; child=child->next) {
if (child->type == XML_ELEMENT_NODE)
{
cout <<child->name<<"\n";
for(xmlAttr* attr=child->properties;attr; attr=attr->next)
{
        xmlNode *subRoot=attr->children;
cout << "Campo:"<<attr->name<<",Valor:"<<subRoot->content<<"\n";
}

}
    }
```



Su ejecución:

**Si fue cargado el archivo**
**Persona**
**Campo:nombre,Valor:Pepe**
**Campo:cedula,Valor:88225**

De manera informal se describen los requerimientos funcionales que debe satisfacer a SIRC para el mecanismo de introspección (RF1 y RF2) y el de realización (RF3 y RF4):

**RF1.** Representar una clase y sus partes.

**RF2.** Consultar la información de la clase, métodos y atributos.

**RF3.** Invocar una clase.

**RF4.** Invocar los métodos de la clase con sus parámetros respectivos.

La explicación de la solución se muestra con un diagrama de casos de uso que permite ilustrar parte de los requerimientos del sistema, véase figura 19.

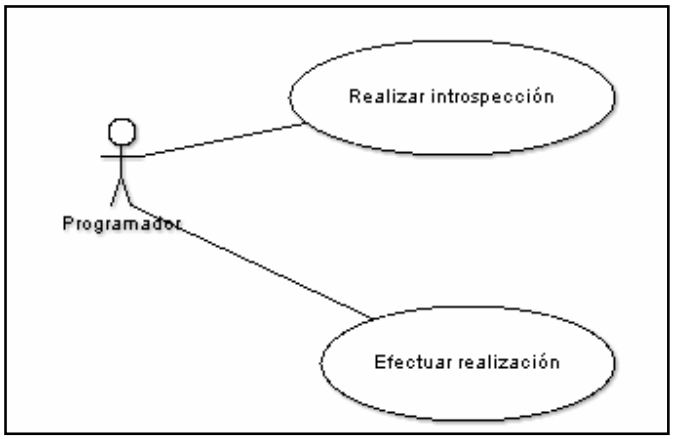



**Figura 19. Diagrama de casos de uso de SIRC**

Las descripciones de los casos de uso se presentan en las tablas 4 y 5.

**Tabla 4. Descripción caso de uso: Realizar instropescción**



| | |
|---|---|
| **Nombre:** | Realizar introspección |
| **Actor:** | Programador |
| **Próposito:**<br>Permite invocar al sistema el mecanismo de introspección. | |
| **Precondiciones:**<br>**1.** El archivo fuente de la clase a reflejar existe.<br>**2.** Se tiene instalado la herramienta gcc-xml<br>**3.** La clase a reflejar no presenta errores de compilación.<br>**4.** Tener la librería libxml2. | |

| |
|---|
| **Flujo Normal:**<br>***1.*** Se toma el fuente y se pasa por gcc-xml de la forma: *gccxml –fxml=File.xml –fxml-start=ClassName sourceClass.cpp*<br>**2.** Se verifica si el archivo se creo en el directorio actual de trabajo.<br>**3.** A través libxml2 se lee el archivo XML generado en el paso 1. Se identifica el elemento *class* y su atributo *members*. El valor del atributo *members* es dividido por espacios blancos, para identificar cada valor de atributos *id* que corresponden a la información de métodos y atributos de la clase reflejada.<br>La información leída es almacenada en el sistema reflexivo. |
| **Poscondición:**<br>El archivo XML es cargado en SIRC. |



**Tabla 5. Descripción caso de uso: Efectuar realización**

| | |
|---|---|
| **Nombre** | Efectuar realización |
| **Actor** | Programador |
| **Descripción:**<br>Permite ejecutar el mecanismo de invocación dinámica de clases. | |
| **Precondición:**<br>**1.** El archivo fuente de la clase a reflejar existe.<br>**2.** La clase a reflejar no presenta errores de compilación. | |
| **Flujo Normal:**<br>**1.** Se obtiene la información de la clase reflejada, de sus métodos y sus atributos.<br>**2.** Se crea el objeto a partir de la clase reflejada. | |

| | |
|---|---|
| | **3.** A través de una interfaz ofrecida por el sistema reflexivo, se invocan los métodos con sus parámetros respectivos. El sistema reflexivo proporciona una función capaz de acceder a los métodos reflejados con su nombre. |
| | **Poscondición**<br>Instancia del objeto reflejado. |

Las figuras 20 y 21 describen los diagramas de actividades para los casos de uso anteriores:

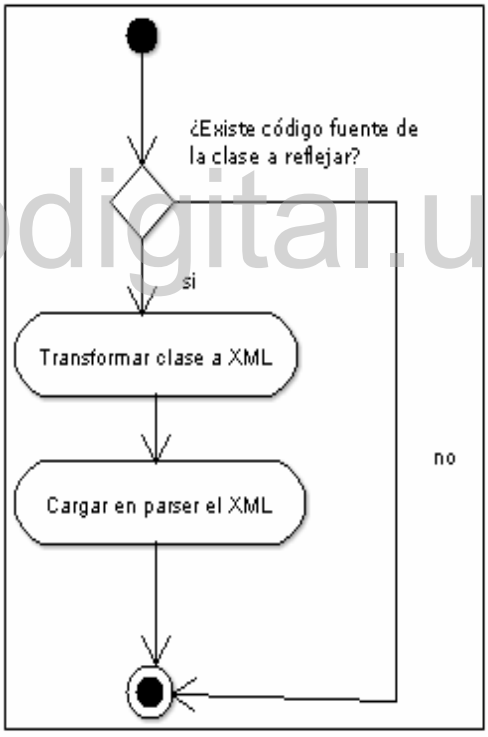



**Figura 20. Diagrama de Actividades: Realizar introspección**

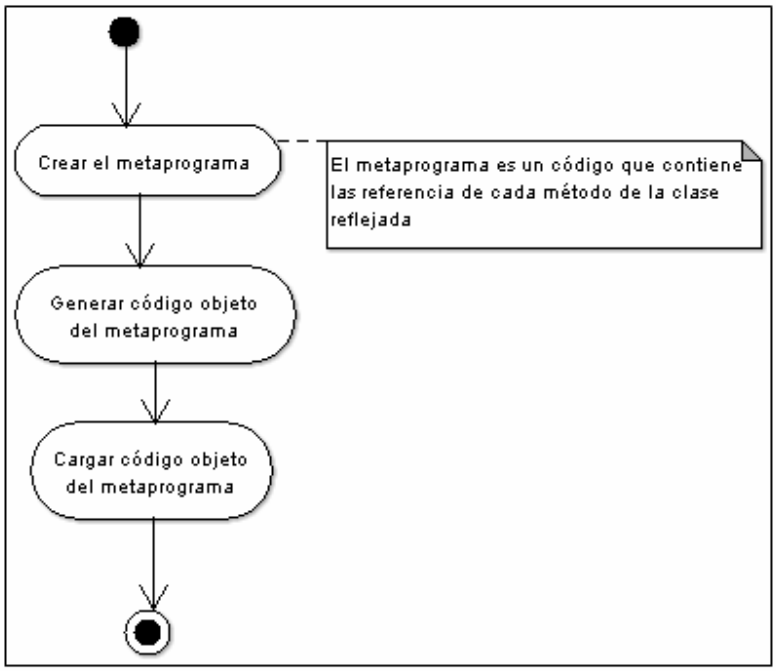



**Figura 21. Diagrama de Actividades: Efectuar realización**



### 4.1.2 Diseño

Las clases de la solución se modelan a partir del concepto formal de una clase: "Una clase es un conjunto que se compone de un tipo particular de metadatos en él que se describen las normas de cómo se comportan los objetos. Una clase tiene una interfaz y una estructura. La interfaz describe los servicios ó métodos que implementan la funcionalidad asociada al objeto y la estructura describe las propiedades o atributos que son características de los objetos"[48].

Se concluye que: Una clase esta compuesta por atributos y métodos. Los métodos dan la funcionalidad de la clase y tienen la misma estructura de una función en un lenguaje estructurado, con excepción de los modificadores de acceso. La figura 22 muestra el diagrama de clase que describe el primer modelo conceptual:

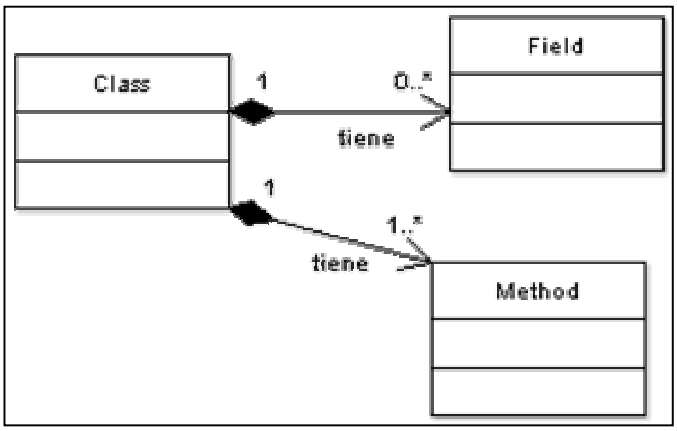



**Figura 22. Modelo conceptual para el caso de uso "Realizar introspección"**

### 4.1.3 Construcción

Mediante la jerarquía de clase presentada en la figura anterior, se procede a la construcción e implementación de la solución. El sistema comienza su funcionamiento tomando como referencia las precondiciones descritas en la tabla 4, que se deben dar como pasos alternos al llamado del sistema. Para evitar este punto se diseña la clase *class,* de tal forma que permita crear el diccionario de forma automática; sin embargo, la carga del código de fuente de la clase se puede realizar a priori a este paso, optimizando el uso del sistema en una aplicación convencional. La clase *class* recibe en uno de sus constructores el nombre y el código fuente de la clase que se desea reflejar.

```
(class.h )
Class::Class(const char *nameClass, const char *source)
{
   std::string fileXML=string(nameClass);
   std::string      order="gccxml      -fxml="+fileXML+"      -fxml-
start="+string(nameClass)+" "+string(source);
   const char *pOrder=order.c_str();
   cout <<"\ncreating dictionary:";
```

```
   system(pOrder);
   cout<<fileXML<<" process completed !!! \n";
   char *FILE=(char*)(fileXML.c_str());
   sourceFile=string(source);
   loadClass(FILE);
}
```

El constructor permite crear el diccionario con la herramienta gcc-xml a través de una llamada al sistema "`system(pOrder)`"; el paso siguiente es cargar el XML comenzando con el elemento <*Class*> que tiene un atributo "id" con valor "_1".

Desde una aplicación de nivel base la invocación típica al sistema sería:

```
char *c=new char[30];
char *source=new char[30];
strcpy(c, "Persona");
strcpy(d, "Persona.h");
Class myClass=Class(c,d);
```



La carga del parser se muestra a continuación:

```
(class.h)
void Class::parser(char *nameClass, char *IDClass)
{
xmlDocPtr document = xmlParseFile(nameClass);
    if (document) {
        ROOT = (xmlNode *) xmlDocGetRootElement(document);
   MetaInfoGenerator(IDClass);
    }
    else {
        cerr << "ERROR XML" << endl;
    }
    xmlCleanupParser();
    xmlFreeDoc(document);
    buildInvokes();
}
```

Las operaciones de carga son similares al procedimiento que se explico en la sección del análisis del problema. `XmlDocPtr` permite cargar el parser de libxml2

y `xmlDocRootElement(…)` obtiene la raíz del árbol de información. Las funciones `xmlCleanupPaser()` y `xmlFreeDoc(…)` permiten liberar el árbol de información referenciado por `xmlDocPtr` y desbloquear el archivo del diccionario. El método `buildInvokes()` tiene la función de crear el metasistema que se irá a explicar en la sección de invocación dinámica.

Con la raíz ya referenciada del árbol de información, se puede dar inicio al proceso de búsqueda y almacenaje en las clases del sistema reflexivo, encontrando cada elemento, y sus atributos, mediante el método `MetaInfoGenerator`. El método recibe el atributo "id" de la clase reflejada y comienza a buscar cada una de sus propiedades. El procedimiento se describe con el siguiente código:

```
(metainformation.h)
void Class::MetaInfoGenerator(char* ID)
{
    xmlNode *rootNode = searchNodeById(ID);
    if (rootNode == NULL) {
        cerr << "No encuentra ID:" << ID << endl;
        return;
    }

    if (strcmp(getTag(rootNode),"Class") != 0) {
        cerr << "No Class" << endl;
        return;
    }
buildClass(rootNode);
}
```



La función `searchNodeById(…)` permite dado el valor del atributo "id" encontrar el elemento cuyo valor de atributo sea igual. La primera actividad será buscar la referencia del elemento <Class> que contiene el atributo "id" con valor igual "_1" que es el que define la estructura de la clase . El método `searchNodeById` recorre los elementos comenzando desde la raíz.

```
(metainformation.h ) ≡
xmlNode* Class::searchNodeById(const char *n)
```

```
{
    if (n == NULL) return NULL;

    xmlNode *child;
    for (child = ROOT->children; child; child=child->next) {
        if (child->type == XML_ELEMENT_NODE &&
            (strcmp(n,matchAttribute(child->properties,"id")) == 0))
            return child;
    }
    return NULL;
}
```

`searchNodeById(…)` realiza una búsqueda sobre cada elemento y para cada uno recorre sus atributo, tarea realizada por el método `matchAttribute(xmlAttr* attr, const char* n)`. Se debe tener presente la estructura de xmlNode que permite tanto encontrar el nombre del elemento como un apuntador a los atributos del nodo procesado.

(metainformation.h)



```
const char* Class::matchAttribute(xmlAttr* attr, const char* n)
{
    if (!attr) return NULL;
    for (xmlAttr* cur = attr; cur; cur = cur->next){
        if (strcmp((const char*)cur->name,n) == 0) {
            xmlNode* value = cur->children;
            assert(value->type == XML_TEXT_NODE);
            return (const char*) value->content;
        }
    }
    return NULL;
}
```

Con la información encontrada del elemento *class* se obtienen los valores del atributo *members* que representa las partes de la clase. El método `buildClass(…)` permite construir la información de métodos y atributos.

A nivel de los métodos se cuentan con una estructura parametrizada con objetos de la clase *method* para modelar los métodos de la clase.

(metainformation.h)

```
void Class::buildClass(xmlNode* n)
{
    name = getAttribute(n,"name");
    char* _id;
    xmlNode* tmp_n;
    if ((_id= const_cast<char*>(getAttribute(n,"bases"))) != NULL){
        _id = strtok(_id," ");
        if ((tmp_n = searchNodeById(_id)) != NULL)
            baseClass = getAttribute(tmp_n,"name");
    }
    char* _members;
   _members = const_cast<char*>(getAttribute(n,"members"));
    _id = strtok(_members, " ");
   int i=0;
    while (_id != NULL) {
        if ((tmp_n = searchNodeById(_id)) != NULL) {
            if (strcmp(getTag(tmp_n), "Field") == 0) {
                fields.push_back(buildField(tmp_n));
            }
            else
        if (strcmp(getTag(tmp_n), "Method") == 0)
                  methods.push_back(buildMethod(tmp_n,false));
        else
        if (strcmp(getTag(tmp_n), "Constructor") == 0)
             constructors.push_back(buildMethod(tmp_n,true));
            }
        _id = strtok(NULL, " ");
        }
    }
```

Obtiene los valores almacenados en <members>

Busca el Elemento



Ya teniendo identificado cada elemento del archivo XML se procede a buscar los campos respectivos de cada parte de la clase. Las operaciones comprenden::

- **Construcción del Atributo:** la construcción del atributo de la clase reflejada se realiza a través del método `buildFields`. El método permite leer la información del atributo *type* del elemento *field* que representa el tipo de dato que puede ser: una clase, la definición de un *struct*, apuntador, vector y demás conjunto de datos abstractos que se pueden crear con la unión de definición de tipos (por ejemplo un apuntador a un vector de objetos); para tales efectos, GCC-XML crea definiciones de elementos con marcas *Pointer_type, Struct, Referente_type* teniendo un comportamiento igual que el de *Field*.

Así por ejemplo si se cuenta con un "int **p" se crearán instancias derivadas <type>(pointer, struct, …) para "int", "*" y "*" .

```
(metainformation.h)
Field*  Class::buildField(xmlNode* n)
 {
            Field *f = new Field();
            f->name = string(getAttribute(n,"name"));
            const char* ch = getAttribute(n,"access");
            f->access = string(ch?ch:"");
            f->type = getTypeName(getAttribute(n,"type"));
            return f;
}
```

- **Construcción del Método y Constructores:** para la construcción del método se realiza a través del elemento <*Method*> que tiene los atributos: *"name", "access", "returns" y "argument"* que representan respectivamente: el nombre del método, su modificador de ámbito, su valor de retorno(si existe) y sus argumentos. Los campos *"returns"* y *"argument"* tienen la misma descripción de tipos realizada por un elemento *field*.



```
(metainformation.h)
Method* Class::buildMethod(xmlNode* node, bool isConstructor)
{
       Method *m = new Method();
       m->name = string(getAttribute(node,"name"));
       const char* ch = getAttribute(node,"access");
       m->access = string(ch?ch:"");
       if(!isConstructor)
            m->ret_type = getTypeName(getAttribute(node,"returns"));
       if (node->children) {
           for (xmlNode* n = node->children; n; n=n->next) {
               if (strcmp(getTag(n),"Argument") == 0)
              {
              std::string
arg_type=getTypeName(getAttribute(n,"type"));
              std::string arg_name=string(getAttribute(n,"name"));
              m->arg_types.push_back(arg_type);
              m->arg_types_name.push_back(arg_type+" "+arg_name);
         }
      }
  }
```

```
    return m;
}
```

### 4.1.4 Realizando consultas

El objetivo principal del sistema es realizar las consultas de las partes de la clase para que el proceso de introspección sea satisfactorio. La tarea mas compleja se realiza en el momento de leer el diccionario y almacenar las clases del sistema. Por lo que los procedimientos de consultas solo se resumen ha realizar una invocación de cada uno de los atributos de la clase *class.* Se realiza el ejemplo basado en un código de prueba inicial, que permite observar las operaciones de la introspección y sus diferentes métodos de consultas (figura 23).

```
1  #include "Reflection/metainformation.hpp"
2  #include <iostream>
3  #include "Persona.cpp"
4  using namespace std;
5  void showInfo(Class myClass);
6
7
8  int main(int argC, char* argV[])
9  {
10
11     char *clase=new char[30];
12     char *fuente=new char[30];
13     strcpy(clase, "Persona");
14     strcpy(fuente,"Persona.cpp");
15     Class myClass=Class(clase, fuente);
16     showInfo(myClass);
17
18 }
```

**Figura 23. Extracto de un código de Prueba: invocación del sistema.**

La función showInfo(…) imprime la información de la clase mostrando el mecanismo de introspección en ejecución (figura 24).

```
40 void showInfo(Class myClass)
41 {
42     vector<Field*> fields=myClass.getFields();
43     vector<Method*> methods=myClass.getMethods();
44     vector<Method*> constructors=myClass.getConstructors();
45     vector<std::string> idClass=myClass.getIdClass();
46
47    cout <<"\n\nMetodos:\n";
48
49    for (vector<Method*>::iterator it = methods.begin(); it != methods.end(); ++it)
50    {
51     cout <<"\tNombre Metodo:\t\t"<<(*it)->getName()<<"\n";
52     cout <<"\tValor Retorno: \t\t"<<(*it)->getRet_type()<<"\n";
53     cout <<"\tModificador de ambito:\t"<<(*it)->getAccess()<<"\n";
54     cout <<"\tArgumentos:\t"<<(*it)->getStringArgumentTypes()<<"\n";
55     cout <<"*****************************************************************************\n";
56    }
57
```

**Figura 24. Extracto de código de prueba: invocación de la introspección**

### 4.2 INVOCACION DINAMICA



Con el diseño de la solución para introspección la tarea final es construir la forma de invocación de la clase reflejada en el sistema reflexivo. La invocación, como se ha tratado en el transcurso del documento, consiste en el llamado de métodos en el mismo sitio o contexto de la clase, sin entrar en mecanismos complejos de invocaciones, como por ejemplo,  casos de métodos heredados.

El mecanismo implementado en este punto supone que la clase cuenta con métodos de obtención y actualización de sus atributos (métodos get y set) y con un constructor vacío; sin embargo, la implantación permite trabajar atributos que sean públicos.

El proceso de reificación en tiempo de ejecución consistirá solo en métodos, aunque por concepto general de "realización" los métodos pueden interactuar con sus atributos e incluso obtener instancias del mismo objeto reificado al nivel base.

### 4.2.1 Análisis del problema

Como se pudo evidenciar en el diagrama de actividades presentado en la figura 21, la invocación dinámica se soporta en los conceptos de "metaprogramación" [23] y en formas de invocación tipo *callback* [25]. La metaprogramación consiste en escribir programas que escriben o manipulan otros programas (o a sí mismos) como datos, o que hacen en tiempo de compilación parte del trabajo que, de otra forma, se haría en tiempo de ejecución. El código no ataca directamente al dominio del problema, sino al código que lo resolvería. Es decir, el código que se escribe no modifica los datos o el estado de la información del programa, sino le proporciona funcionalidades.

*Callback* es una técnica que permite que un código ejecutable pueda ser pasado como un argumento de otro código, permitiendo que un programa ejecute segmentos de código de otro. En el caso de C++ el concepto se implementa con apuntadores a funciones o a métodos [27]. Por lo general, debe existir un código que manipula y tiene las definiciones de los punteros que manejan las direcciones de los métodos y/o funciones (figura 25):



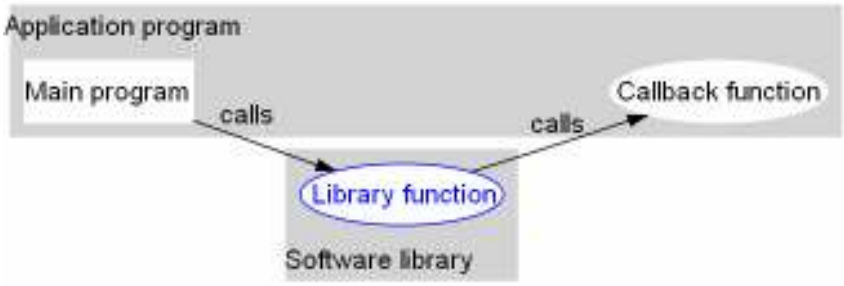



**Figura 25. *Callback*** (tomado de [26])

Para definir un apuntador a un método o función se usa la siguiente sintaxis:

- Función: tipo (* apuntador) (parámetros,…).
- Método: tipo (Clase_que_contiene_método::*apuntador) (parámetros,…).

Para asignar una dirección:

- Función: apuntador=&Función.
- Método: apuntador=&Clase_que_contiene_método::método.

Por razones de compatibilidad de tipos la asignación de direcciones es estricta y solo acepta métodos y/o funciones que tengan la misma firma [24][27]. Para efectos de invocación (si es de tipo *void* la función o el método solo se realiza el llamado de la forma convencional):

- Función: tipo dato_retorno=(*apuntador)(valores,…).
- Método: tipo dato_retorno=(objeto.*apuntador) (valores,…).

A continuación se presenta un ejemplo de llamado de métodos a través de otra clase usando apuntadores a métodos. El llamado se realiza con los métodos *"invoke1"* e método *"invoke2"* de la clase *callback;* que referencian los métodos getNombre() y *getAlgo(…)* de la clase persona, respectivamente (figura 26):



```
1  #include "Persona.h"
2  class callBack{
3  std::string invoke1()
4  {
5  Persona *object=(Persona *)obj;
6  std::string(Persona::*pt_getNombre)();
7  pt_getNombre=&Persona::getNombre;
8  (std::string result=(object->*pt_getNombre)();
9  return (result);
10 }
11
12 int invoke2(int a, float b, char c, long d)
13 {
14 int(Persona::*pt_getAlgo)(int,float,char,long int);
15         pt_getAlgo=&Persona::getAlgo;
16         int result=(object->*pt_getAlgo)(a,b,c,d);
17 return (result);
18 }
19
20 } //Fin class CallBack
```

**Figura 26. Un ejemplo de *callback***

De la figura 26, las líneas 5 a la 8 y de la 14 a la 16 presentan la definición y la invocación de métodos que pertenecen a la clase persona, a través de la clase de

*callback*. Si se generaliza lo presentado en el ejemplo, y se quisiera automatizar él llamado y realizar un sola operación que pueda invocar todos los métodos de la clase, la información que se necesitaría conocer sería:

1. Nombre del método.
2. Tipo de dato del valor de retorno, si existe.
3. Cantidad de parámetros con sus tipos de datos, si existen.

El problema radica en implementar un método que reciba una cantidad variable de parámetros de diferentes tipos de datos, así como, almacenar y retornar un valor si el método lo exigiera. En C++ la representación de datos se puede realizar con apuntadores de tipo *void* que permiten almacenar la dirección de memoria de cualquier tipo de dato. La clase de la figura 26 se puede reescribir en el código que se muestra en la figura 27:



**Figura 27. Clase *callback* implementada con apuntadores void**

De la figura 27, la línea 3 define los argumentos:

1. void *obj: corresponde a cualquier objeto de la clase persona.
2. string name: nombre del método.

3. void *result: variable que almacena el resultado; esto es, si el método tiene un valor un retorno.
4. void *args[]: parámetros que reciben los métodos.

El proceso amerita una comprensión a nivel de la interpretación y cambio de tipos de datos. La operación implementada anteriormente permite evidenciar la solución en la tarea de "realización" que implementa SIRC. Se podría crear una clase de forma dinámica como la *callback* con la definición de métodos y tipos.

### 4.2.2 Diseño

Mediante el mecanismo de introspección implementado, La solución ahondó en desarrollar formas que permitiesen realizar la invocación de un objeto perteneciente a la clase reflejada y sus métodos. Como se explico anteriormente, se hizo uso de la metaprogramación, la cuál permitió generar el código necesario para producir invocaciones a través de punteros a métodos.



El mecanismo de invocación debe de alguna manera crear y cargar los objetos del nivel meta. El programa que almacena los llamados a los métodos se denominará metaprograma. La solución se dividió en analizar los siguientes puntos:

1. Construcción del metaprograma: código fuente.
2. Creación del código objeto a partir del metaprograma.
3. Diseño de la clase que invoca el código del metaprograma. Este punto es de relevancia para realizar el acoplé en cuanto a diseño e implementación de SIRC con su mecanismo de introspección: mediador.

#### 4.2.1.1 Nivel meta

El nivel meta contiene los objetos reflejados que permiten que el nivel base pueda invocar los servicios ofrecidos por cada uno de los metaobjetos. Por esta razón el

metaprograma debe contener el código necesario para realizar las invocaciones respectivas. Esto lleva a crear el código después de un proceso de introspección donde se conoce la metainformación de la clase. El metaprograma sirve de mediador entre el nivel base y el meta, contiene los llamados o invocaciones de cada uno de los métodos de la clase reflejada. El soporte es proporcionado a través de las clases *"Class"* y *"Method"* que permiten construir los segmentos de código del metaprograma. Por parte del método se crean cuerpos de código de tipo:

```
if (nombre_método="algún método")
{
   1. Declaración de apuntador a clase reflejada;
   2.  Apuntador = Dirección del método de la clase reflejada;
   3.  Proceso de Invocación del método con sus argumentos.
}
```

En la clase *Method* se diseño un método de nombre "`getBodyInvokes(…)`" que tiene la función de crear para cada método "publico", un segmento de código para el metaprograma, su código es presentado en la figura 28:



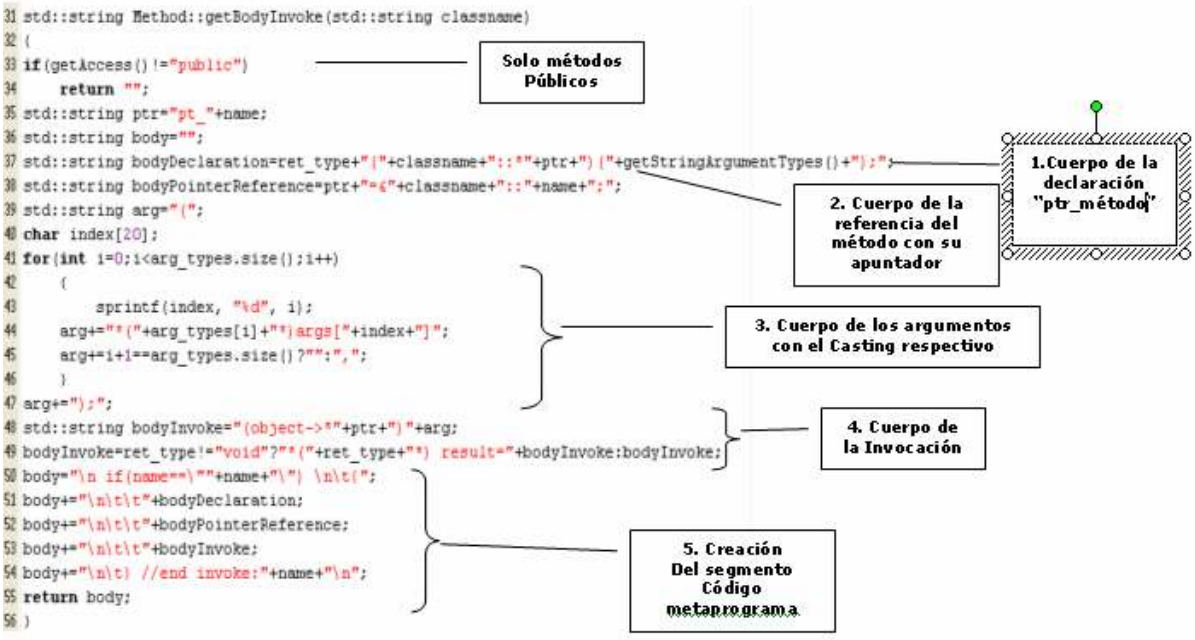



**Figura 28. Fragmento de código de la clase "Method" que permite la construcción de invocaciones**

La operación del método se basa en la información de la clase ya cargada en el sistema reflexivo, si se toma la clase de ejemplo persona, el código resultante es igual a lo planteado en el ejemplo de *"callback"* (figura 27) con los pasos descritos en la figura anterior se muestra un segmento de código que referencia a un método de la clase de ejemplo (figura 29):

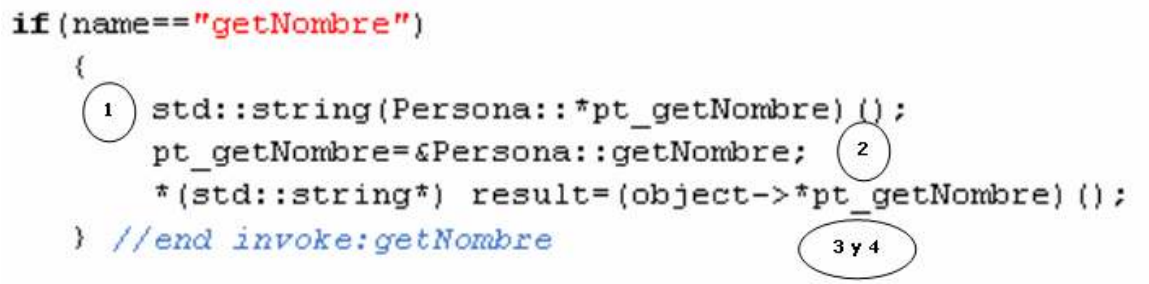



**Figura 29. Segmento de Código de un metaprograma**



Con la generación del código para cada método, solo resta la operación de invocarlo parte de la clase *Class* que contiene una colección (`std::vector<Method *>`) de objetos de tipo *Method.* El código es presentado a continuación:

```
(Class.h )
void Class::buildInvokes()
{
   1. std::string header="#include \""+sourceFile+"\" \n";
   2. std::string castingObj=name+" *object=("+name+"*)obj;";
   3. std::string filename=getName()+"_metaProgram.cpp";
   4. const char *FILE=filename.c_str();
   5. ofstream fs(FILE);
   6. std::string bodyMethod, bodyField;
   7. bodyMethod="\n{\n"+castingObj+"\n";
   8. for(int i=0;i<methods.size();i++)
            i. bodyMethod+=methods[i]->getBodyInvoke(name);
   9. bodyMethod+="\n }\n";
   10.     fs<<INVOKEMETHOD;
   11.     fs<<bodyMethod;
   12.     fs.close();
   13.     std::string          order="g++          -shared          -o
      "+getName()+"metaProgram.so"+" "+filename;
   14.     const char *pOrder=order.c_str();
   15.     cout <<"\ncreating metasystem:";
   16.     system(pOrder);
```

```
17.     cout<<getName()+".so  process completed !!!\n";
18.     }
```

Las Líneas 1 y 2 almacenan respectivamente cadenas que representan: la librería que contiene el código fuente (previamente leída en el constructor de la clase "*Class*") y la cadena que permite el *casting* del objeto reflejado que produce la invocación de cada uno de los métodos. No se debe olvidar que se esta creando el metaprograma, por lo que este método solo opera con variables que almacenan caracteres que van a permitir formatear su código. Las líneas 8 y 9 se formatean el cuerpo de la función. Las líneas 10 a la 12 crean el archivo y lo colocan en el directorio actual de trabajo; el resto de líneas pertenecen a la operación de la carga del metaprograma que se estudiará en la siguiente sección. El método crea un archivo de nombre "Clase_metaPrograma.cpp" y genera el código objeto ".Clase_metaPrograma.so".



#### 4.2.1.2 Mecanismo de carga del metaprograma

Después de creado el código del metaprograma la tarea siguiente es encontrar el mecanismo que permita cargar el código para realizar las diferentes invocaciones.

La implementación no podía estar ajena al diseño del sistema reflexivo propuesto; razón por la cuál el diagrama de clase presentado en la figura 20 se le añade una clase que representa el objeto reflejado.

La solución modela formalmente el comportamiento de una clase donde se crea un *objeto* y a través del "objeto reflejado" se tienen invocaciones a sus métodos. Por efectos de complejidad en la solución, no se realiza reificación de métodos estáticos que pueden ser invocados sin la creación del objeto.

La clase *Object* tendría el comportamiento general de un *singleton* [28][29] con la adición de que el constructor de la clase recibiría como parámetro un objeto de tipo *Class* que permite invocar la creación del metaprograma.

Se toma como referente el patrón *singleton,* pues se diseña para restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto. Su intención consiste en garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella. El patrón *singleton* [31] provee una única instancia global gracias a que: la propia clase es responsable de crear la única instancia y permite el acceso global a dicha instancia mediante un método de clase (figura 30).

**Figura 30. Diagrama de clase del patrón singleton (tomado de [30])**



En el caso de C++ la solución es presentada por un diseño conocido como el singleton de Mayer [29][32]:

```
template<typename T> class Singleton
{
    public:
        static T& Instance()
        {
            //asumir T posee un constructor por defecto
            static T laInstanciaSingleton;
            return laInstanciaSingleton;
        }
};
```

Tomando como referencia el diseño anteriormente expuesto y el modelo conceptual de SIRC presentado en el mecanismo de introspección, se muestra la propuesta final del diagrama de clase de SIRC (figura 31).

Las operaciones de la carga dinámica del metaprograma y de la interfaz para permitir invocaciones a métodos se encapsulan en una clase de nombre *Object*.

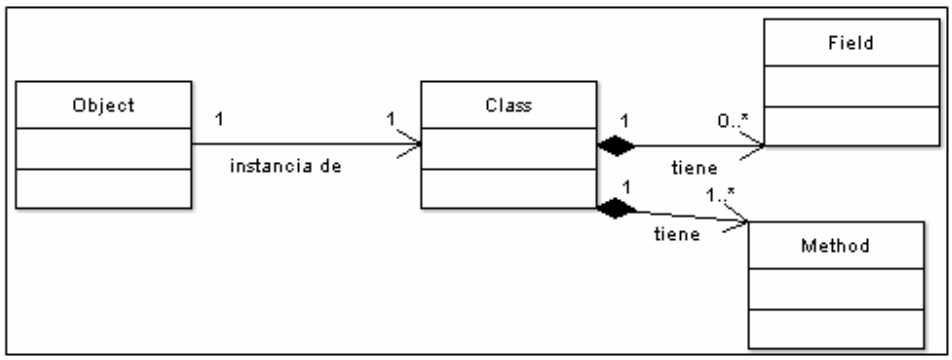



**Figura 31. Modelo Conceptual de SIRC**

Para la creación del objeto en forma dinámica la clase *Object* se parametriza de tal forma que se pueda trabajar con las características que ofrece los *templates*. El proceso de invocación de los métodos del objeto referente se realiza a través de apuntadores a los métodos de la clase reflejada.



C++ ofrece un conjunto de herramientas que permiten crear un enlace dinámico a una librería que carga un programa en tiempo de ejecución. En lugar de ser enlazadas en tiempo de compilación, se mantienen como archivos independientes separados del fichero ejecutable del programa principal. La mayor parte de la labor de enlazado se realiza en el momento en que la aplicación se carga o durante la ejecución.

El metaprograma contiene el código objeto que se va a ser enlazado con la aplicación que se tenga en el nivel base. El mecanismo se acciona cuando el programa accede a sus funciones por primera vez. El objetivo de la clase *Object* es cargar dinámicamente el metaprograma y conceder una interfaz al usuario para invocar los métodos del objeto referente.

La carga dinámica se realizó a través de un conjunto de funciones denominadas *Dynamic Loader Compatibility* (DLC)[16] [33] [34] que permiten a través de cinco funciones definidas en el fichero "dlfcn.h" realizar la carga de forma transparente e independiente de la plataforma. Sus funciones se describen a continuación:

- `dlopen()`: abre una librería de enlace dinámico. Una aplicación debe llamar a esta función antes de usar cualquier símbolo exportado de la librería. Si la librería no está abierta por el proceso llamante, la librería se carga en la memoria del proceso.
- `dlclose()`: se usa por el proceso para cerrar la conexión con la librería.
- `dlsym()`: retorna la dirección de un símbolo exportado por una librería de enlace dinámico.
- `dladdr()`: recibe una dirección de memoria y , si esta corresponde con la dirección de memoria de una variable o función de la librería , devuelve información sobre este símbolo.
- `dlerror()`: devuelve una cadena con la descripción del error producido en la última llamada a `dlopen()`, `dlsym()` o `dladdr()`.



Para usar las instrucciones del DLC la cabecera de la función debe formatearse con la cadena `extern 'C'` cuya función es indicar al compilador que la declaración será accesible desde otras unidades de compilación [35].

### 4.2.3 Construcción

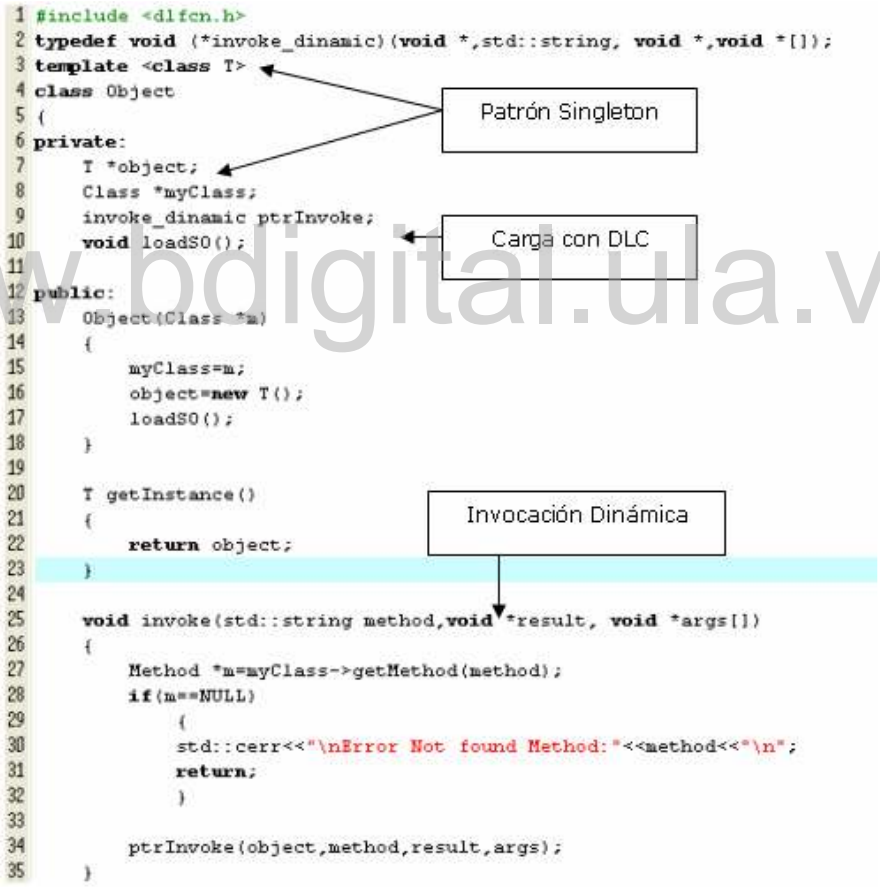
La creación del metaprograma se realiza a través del método `buildInvokes()` de la clase *Class*, que permite crear el código necesario para la construcción de la librería dinámica; el método permite, además, generar el código objeto con una llamada al sistema con la orden del compilador:

[16] Conjunto de funciones creadas por Jorge Acereda y Peter O'Gorman con el objetivo de permitir la portabilidad y carga de las librerías.

```
"g++ -shared -o "+getName()+"metaProgram.so"+" "+filename;
```

La opción "`-shared`" del compilador, le indica que se va a crear una unidad de código que va a ser usado desde otro programa..

La función de la clase *Object* es invocar, las funciones DLC, para cargar la unidad de código objeto que representa el metaprograma, que contiene las referencias a los métodos de la clase reflejada. El código fuente de la clase *Object* se presenta en la figura 32.




```
1  #include <dlfcn.h>
2  typedef void (*invoke_dinamic)(void *,std::string, void *,void *[]);
3  template <class T>
4  class Object
5  {
6  private:
7      T *object;
8      Class *myClass;
9      invoke_dinamic ptrInvoke;
10     void loadSO();
11
12 public:
13     Object(Class *m)
14     {
15         myClass=m;
16         object=new T();
17         loadSO();
18     }
19
20     T getInstance()
21     {
22         return object;
23     }
24
25     void invoke(std::string method,void *result, void *args[])
26     {
27         Method *m=myClass->getMethod(method);
28         if(m==NULL)
29             {
30             std::cerr<<"\nError Not found Method:"<<method<<"\n";
31             return;
32             }
33
34         ptrInvoke(object,method,result,args);
35     }
```

**Figura 32. Clase Object**

El método `loadSO` (figura 33) de la clase *Object*, permite cargar e inicializar el apuntador a la función `invoke` que se tiene en el metaprograma.

```
void loadSO()
{
    std::string loadFile="./"+myClass->getName()+"metaProgram.so";
    const char *lF=loadFile.c_str();
    void *ptr = dlopen(lF, RTLD_LAZY);
    if(!ptr)
        cout<<dlerror()<<"\n";
        dlerror();
        ptrInvoke= (invoke_dinamic) dlsym(ptr, "invoke");
        const char *dlsym_error = dlerror();
        if (dlsym_error)
        {
        cerr << "Cannot load symbol 'invoke': "<<dlsym_error<<"\n";
        dlclose(ptr);
            return;
            }

}
```

**Figura 33. Método de la clase *object* que permite cargar el metaprograma**

A través del método `invoke` de la clase *Object* se realizan las invocaciones a los métodos de la clase reflejada. *Object* recibe en su constructor una referencia a un objeto tipo *Class,* que permite invocar el mecanismo de introspección para completar la operación de realización. Complementando el ejemplo de la figura 23, se muestra una llamada típica al sistema reflexivo (figura 34) con los mecanismos de introspección y realización en funcionamiento.



```
 8 int main(int argC, char* argV[])
 9 {
10
11     char *clase=new char[30];
12     char *fuente=new char[30];
13     strcpy(clase, "Persona");
14     strcpy(fuente,"Persona.cpp");
15     Class myClass=Class(clase, fuente);
16     showInfo(myClass);
17     std::string rta;
18     Object<Persona> obj(&myClass);
19     obj.invoke("getNombre", &rta,NULL);
20     cout <<"Resultado de invocar getNombre:"<<rta<<"\n";
21     void *p[5];
22     int a1=34;
23     float a2=3.4;
24     char a3='p';
25     long a4=234;
26     p[0]=&a1, p[1]=&a2,p[2]=&a3, p[4]=&a4;
27     int rta2;
28     obj.invoke("getAlgo",&rta2, p);
29     cout <<"Resultado de invocar getAlgo:"<<rta2<<"\n";
30     obj.invoke("getAlgoss",&rta2, p);
31     return 0;
32 }
```

**Figura 34. Prueba del proceso de Invocación**

De la figura 34 se presenta:

- Proceso de introspección: líneas 15 y 16.
- Proceso de Invocación: la invocación se realiza con la clase *object*. La línea 18, presenta la clase parametrizada con el objeto reflejado, que recibe en su constructor la metainformación representada por un objeto de la clase *class*.
  Las invocaciones se realizan a través del método `invoke`, que es una interfaz entre *object* y el metaprograma. El método recibe como parámetros el nombre del método reflejado, la referencia al dato que va almacenar valor de retorno (si existe) y los parámetros en un arreglo de apuntadores `void`. En el caso del segundo método "`getAlgo`" posee parámetros, por lo que, se deben encapsular en un vector tipo `void` (línea 21 a la 26). La línea 30 genera un error, y el sistema lo reporta ya que el método "`getAlgoss`" no existe en la clase persona.

## 4.4 TAXONOMIA DEL SISTEMA.



La solución esta dividida en las siguientes librerías y componentes externos que se utilizaron para dar soporte a la implementación (figura 35):

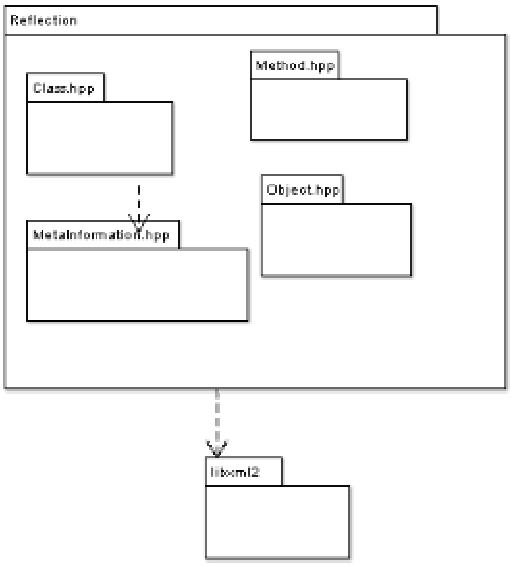



**Figura 35. Taxonomía de SIRC**

Como se observa en la figura anterior es necesario tener instalado el toolkit libXML2 y redireccionado al /usr/lib del sistema, las instrucciones de instalación se encuentran en el INSTALL del CD anexo al documento.

# 5. CONCLUSIONES Y RECOMENDACIONES

## 5.1 CONCLUSIONES

Durante el desarrollo del trabajo se identificó que las implementaciones de reflexión ofrecidas para C++ realizan sus procesos de consultas y de invocaciones por etapas que requieren colocación de código invasivo; por lo tanto, el programador tiene que escribir casi en su totalidad el código fuente de una clase. Hecho que ha generado que el desarrollo de aplicaciones reflexivas en C++ sea escaso.

Por otra parte, gran cantidad de la literatura a cerca de la reflexión, esta enfocada a la programación funcional y solo algunos autores han adaptado los conceptos a la OO de manera general y a la descripción de implantaciones en ciertos lenguajes.



El aporte principal del trabajo es una base teórica formal de la reflexión en la OO y el uso de estos conceptos en el diseño y construcción de SIRC. La arquitectura reflexiva orientada a objetos desarrollada permite: invocar las operaciones de introspección y realización de forma transparente al programador, una API que ofrece la posibilidad de cargar una jerarquía de clases y la obtención de instancias de objetos reflejados con métodos de una cantidad de parámetros variable. El diseño de la librería brinda una solución que evita la intromisión de código y las etapas de precompilación; razón por la cuál, la integración con otros sistemas resulta más sencilla.

SIRC ofrece un mecanismo reflexivo con reificación estructural y de computación. El primero especifica una estructura de clases para su consulta y el segundo se encarga de la invocación de un método. La reificación convierte una cadena que corresponde al nombre simbólico de una clase o método en una referencia que permite su manipulación e invocación, similar a los lenguajes que ofrecen reflexión

de forma nativa, lo que facilita la adaptabilidad y reusabilidad del código fuente de la clase reflejada en cualquier aplicación.

Se logró comprobar que aunque C++ no posee un soporte de reflexión propio, si es posible otorgar un mecanismo de reflexión completo. Lo crítico era encontrar las técnicas y herramientas que permitieran la lectura y manipulación de la información de la clase.

Con la idea ofrecida por *Reflex* para la representación de la clase en un XML, los toolkits GCC-XML y libXML2 se desarrolla el soporte de introspección. El concepto de metaprogramación (código que escribe código) fué la base para la invocación dinámica que se realiza a través de la creación y carga de una pieza de código objeto en tiempo de ejecución dejando atrás las etapas de precompilación de las soluciones actuales.



## 5.2 RECOMENDACIONES

La solución sólo basa sus operaciones a nivel de intra proceso; por lo que un trabajo futuro comprendería la ampliación del soporte reflexivo en un ambiente distribuido. Desde SIRC, por ejemplo, se podría adaptar su API para realizar carga de repositorios de clase, el problema se enfocaría en buscar los medios para entrar en un espacio distribuido, serializar el objeto e invocarlo desde cualquier sitio.

De igual forma, surgen inquietudes de cuando es posible usar un enfoque reflexivo y como integrar las soluciones a nivel de ingeniera de software y la construcción del código. Se hace necesario el estudio de metodologías para el desarrollo de aplicaciones adaptativas, tal como lo presenta un trabajo realizado por Cazzola [53]. En él se describe un modelo para la construcción y adaptación de un sistema en función de los requerimientos no funcionales.

Finalmente, con respecto a SIRC se recomienda lo siguiente:

- Implementar un servicio sobre la clase *Object* que permita la invocación de métodos estáticos. Para esto es necesario codificar un método que a través del mecanismo de introspección genere el código en el metaprograma para obtener la referencia hacia los métodos estáticos de la clase reflejada.
- Incorporar un servicio que permita invocación y reconocimiento de métodos sobrecargados. Para esto es necesario buscar una estrategia para la identificación exacta de los tipos de datos de los parámetros.
- Permitir al usuario añadir código de métodos y atributos a la clase reflejada.

## REFERENCIAS BIBLIOGRAFICAS


1. MAES, Pattie. Concepts and experiments in computation reflection. http://www.itu.dk/people/ydi/PhD_courses/adaptability_design/PattieMaesOOPSLA87.pdf. (Citado en 24 de septiembre de 2007)

2. ORTÍN, Francisco. "Sistema Computacional de Programación Flexible diseñado sobre una Máquina Abstracta Reflectiva no Restrictiva". Tesis Doctoral. Universidad de Oviedo. Diciembre 2001. http://www.tesisenxarxa.net/TDR-0425107-094512/. (Citado en 24 de septiembre de 2007)

3. BRIAN, Foote. Reflective Facilities in Smalltalk-80. http://www.laputan.org/ref89/ref89.html. (Citado en 24 de septiembre de 2007)

4. DEMERS Frederick, MALENFANT Jackes. Reflection in logic, Functional and object oriented programming. http://www.itu.dk/people/ydi/PhD_courses/adaptability_design/DemersMalenfant95.pdf. (Citado en 24 de septiembre de 2007)

5. MARKWITZ, Victor. Data Management Research and Development Group Information and Computing Sciences Division Lawrence Berkeley Laboratory. 1993. The Object Protocol Model. Draft 1.5. I –Min A Chen. http://citeseer.ist.psu.edu/cache/papers/cs/350/ftp:zSzzSzftp.informatik.uni-stuttgart.dezSzpubzSzdkfzzSzOPM.pdf/chen93objectprotocol.pdf. (Citado en 04 de noviembre de 2007)

6. DUCASSE, S. A reflective model for first class dependencies. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.39.306. (Citado en 04 de noviembre de 2007)

7. KICZALES, G. The Art of the Metaobject Protocol. http://citeseerx.ist.psu.edu/showciting?cid=63564. (Citado en 10 de noviembre de 2007)

8. HERMANPREET, Singh. Introspective C++. http://scholar.lib.vt.edu/theses/available/etd-11292004-155755/unrestricted/index.pdf. (Citado en 10 de noviembre de 2007)

9. TYNG-RUEY, Chiang. Non-intrusive object introspection in C++. http://www.iis.sinica.edu.tw/ ~trc/spe-introspection.pdf. (Citado en 10 de noviembre de 2007)

10. LESIECKI, Nicholas. Improve modularity with aspect-oriented programming. http://www-128.ibm.com/developerworks/java/library/j-aspectj/. (Citado en 10 de noviembre de 2007)

11. MERTZ, David. A Primer on Python Metaclass Programming. http://www.onlamp.com/lpt/a/3388. (Citado en 12 de noviembre de 2007)

12. COOPER, Richard, Type-Safe Linguistic Run-time Reflection Type-Safe ftp://ftp.dcs.gla.ac.uk/pub/fide/reports/fide_94_108.ps.gz. (Citado en 09 de diciembre de 2007)

13. KALEV, Danny. Use RTTI for Dynamic Type Identification. http://www.devx.com/getHelpOn/Article/10202. (Citado en 14 de diciembre de 2007)

14. CHIBA, Shigeru. A Metaobject Protocol for C++. http://www.csg.is.titech.ac.jp/~chiba/pub/chiba-oopsla95.ps.gz. (Citado en 18 de diciembre de 2007)

15. STROUSTRUP, Bjarne. The C++ Programming Language. Tercera edición. Addison-Wesley. ISBN 0-201-88954-4

16. ROISER, S. The SEAL C++ Reflection System. In Computing in High Energy and Nuclear Physics (CHEP), September 2004. (Citado en 15 de diciembre de 2007)

17. THE WORLD WIDE WEB CONSORTIUM. Extensible Markup Language (XML) 1.0. http://www.w3.org/TR/REC-xml/ . (Citado en 20 de diciembre de 2007)

18. WIKIPEDIA. Metalenguaje. http://es.wikipedia.org/wiki/Metalenguaje. (Citado en 20 de diciembre de 2007)

19. THE WORLD WIDE WEB CONSORTIUM. Document Object Model (DOM). http://www.w3.org/DOM/ . (Citado en 20 de diciembre de 2007)

20. GNOME. The XML C parser and toolkit of Gnome . http://xmlsoft.org/ . (Citado en 20 de diciembre de 2007)

21. THE WORLD WIDE WEB CONSORTIUM. XML Path Language (XPath) Version 1.0. W3C Recommendation 16 November 1999. http://www.w3.org/TR/xpath. (Citado en 21 de diciembre de 2007)

22. MILOSLAV, Nic y JIRAT, Jiri. XPath Tutorial. http://www.zvon.org/xxl/XPathTutorial/General/examples.html . (Citado en 21 de diciembre de 2007)

23. BARTLETT, Jonathan. The art of metaprogramming. http://www-128.ibm.com/developerworks/linux/library/l-metaprog1.html?ca=dgr-wikiaMetaprogramming. (Citado en 05 de enero de 2008)

24. HAENDEL, Lars. The Function Pointer Tutorials. http://www.newty.de/fpt/index.html. (Citado en 05 de enero de 2008)

25. JAKUBIK, Paul. Callback Implementations in C++. http://www.newty.de/jakubik/callback.pdf. (Citado en 08 de enero de 2008)

26. WIKIPEDIA. Callback (Computer Science) . http://en.wikipedia.org/wiki/Image:Callback-notitle.svg. (Citado en 08 de enero de 2008)

27. HICKEY, R. "Callbacks in C++ Using Template Functors", C++ REPORT, 7(2), 1995. (Citado en 08 de enero de 2008)

28. GAMMA, Erich. Design Patterns: Elements of Reusable Object-Oriented Software. 1994. ISBN-13: 978-0-201-63361-0.

29. MEYERS, Scott. C++ and the Perils of Double-Checked Locking. http://www.aristeia.com/-Papers/DDJ_Jul_Aug_2004_revised.pdf#search=%22meyers%20double%20checked%20locking%22. (Citado en 05 de enero de 2008)

30. GEARY, David. Simply Singleton. http://www.javaworld.com/javaworld/jw-04-2003/jw-0425-designpatterns.html. (Citado en 05 de enero de 2008).



31. WIKIPEDIA. UML class diagram for Singleton software design pattern. http://upload.wikimedia.org/wikipedia/commons/f/fb/Singleton_UML_class_diagram.svg. (Citado en 05 de enero de 2008)

32. COPLIEN, James O. Curiously Recurring Template Patterns. http://en.wikipedia.org/wiki/Curiously_Recurring_Template_Pattern. (Citado en 08 de enero de 2008)

33. NORTON, James. Dinamyc Class Loading for C++ on Linux. http://www.linuxjournal.com/article/3687 . (Citado en 08 de enero de 2008)

34. ISOTTON A. C++ dlopen mini HOWTO. http://www.isotton.com/howtos/C++-dlopen-mini-HOWTO. (Citado en 08 de enero de 2008)

35. STROUSTRUP, Bjarne. The C++ Programming Language. http://www.research.att.com/~bs/. (Citado en 12 de enero de 2008)

36. SHIGERU, Chiba y MASUDA Takashi. Designing an Extensible Distributed Language with a Meta-Level Architecture. http://www.csg.is.titech.ac.jp/~chiba/pub/chiba-ecoop93.ps.gz. (Citado en 13 de enero de 2008)

37.SOULIE, Juan. Templates in c++, http://www.cplusplus.com/doc/tutorial/templates.html. (Citado en 15 de enero de 2008)

38. MYERS, Glenford J, The Art of Software Testing, John Wiley & Sons inc, 2004.

39.HOHMUTH , Michael .A preprocessor for C and C++ modules. http://os.inf.tu-dresden.de/~hohmuth/prj/preprocess/. (Citado en 15 de enero de 2008)

40.VOLLMANN, Detlef. Metaclasses and Reflection in C++. http://www.vollmann.com/pubs/meta/meta/meta.html. (Citado en 20 de enero de 2008)

41.EUROPEAN ORGANIZATION FOR NUCLEAR RESEARCH (CERN). *Reflex* - Reflection for C++. seal-reflex.web.cern.ch/seal-reflex/. (Citado en 20 de enero de 2008)

42.SMITH, Brian. Reflection and Semantics in Lisp. http://portal.acm.org/citation.cfm?id=800513. (Citado en 10 de febrero de 2008)

43.SOBEL, Jonathan M. An Introduction to Reflection-Oriented Programming. http://www2.parc.com/csl/groups/sda/projects/reflection96/docs/sobel/rop.html. (Citado en 17 de febrero de 2008)



44.RAO R, Implementational Reflection in Silica. http://www.itu.dk/people/ydi/PhD_courses/adaptability_design/Rao91Implementation.pdf. (Citado en 20 de febrero de 2008)

45.GREGOR, Kiczales. An Overview of AspectJ. http://www.parc.com/research/projects/aspectj/downloads/ECOOP2001-Overview.pdf. (Citado en 05 de marzo de 2008)

46.APPLE. Introduction to The Objective-C 2.0 Programming Language. http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/index.html. (Citado en 05 de marzo de 2008)

47.PROYECTO GCC-XML. http://www.gccxml.org/. (Citado en 05 de marzo de 2008)

48.MEYER, B. Object-oriented software construction, 2nd edition, Prentice Hall, 1997. (Citado en 05 de marzo de 2008)

49.SOSNOSKI, Dennis. Java programming dynamics. http://www.ibm.com/developerworks/library/j-dyn0603/. (Citado en 05 de marzo de 2008)

50. SUN MICROSYSTEMS. Java Core Reflection, http://java.sun.com/j2se/1.4.2/docs/guide/reflection/spec/java-reflection.doc.html#8589. (Citado en 15 de marzo de 2008)

51. WHALEY John, LIVSHITS Benjamín y LAM Monica. Reflection Analysis for Java. http://suif.stanford.edu/papers/aplas05r.pdf. (Citado en 22 de marzo de 2008)

52. BARCZIKAY, Peter. Advanced Run Time Type Identification in C++. http://www.rcs.hu/Articles/RTTI_Part1.htm. (Citado en 22 de marzo de 2008)

53. *CAZZOLA, Walter.* Reflective Analysis and Design *for Adapting Object Run Time Behavior.* www.disi.unige.it/person/CazzolaW/ps/oois2002.pdf. (Citado en 05 de abril de 2008)