Desarrollo de Aplicaciones C++ Modificables o Extensibles en Tiempo de Ejecución

Oscar Antezana Chávez, Gonzalo Argote García

Instituto de Investigación en Informática Aplicada
Universidad Católica Boliviana - Regional Cochabamba
e-mail: antezano@ucbcba.edu.bo

Resumen

Todo sistema de software es dinámico y sufre durante su ciclo de vida diversos cambios producto del avance tecnológico, cambios en las funcionalidades presentes o extensiones para el soporte de nuevas funcionalidades. De forma tradicional, las modificaciones a un sistema de software las realizan los programadores, obteniendo una nueva versión del mismo. Para poner en funcionamiento esta nueva versión se requiere detener la ejecución del software antiguo, reemplazarlo por el nuevo e iniciar la ejecución con el nuevo software. En este artículo se presenta una técnica que permite desarrollar software con la capacidad de ser modificado mientras está en ejecución. Este software, que será denominado *Sistema Dinámicamente Extensible y Modificable*, está construido sobre la base de una arquitectura de capas, cuyos subsistemas más importantes están implementados en C++ para el entorno Windows 95 y superiores.

Palabras Clave: C++, programación orientada a objetos, enlace dinámico, extensión de aplicaciones, diseño de sistemas.

1 Introducción

"Planeado o no, el software tiende a evolucionar con el tiempo"[2]. Lamentablemente, los cambios requeridos para modificar un programa ya en funcionamiento deben ser realizados por el grupo de desarrolladores, quienes, una vez terminados los cambios, deben actualizar el antiguo sistema[1]. Este proceso de actualización, generalmente se realiza deteniendo el sistema original, reemplazándolo por la nueva versión e iniciando la ejecución de dicha nueva versión.

El detener la ejecución de un sistema puede resultar inviable en algunos casos, como en aquellos donde los sistemas realizan actividades críticas tales como el control del

[1]En este artículo, *sistema* siempre se refiere a *sistema de software*.

tráfico aéreo, control de procesos industriales, etc [8, 6]. En este tipo de sistemas, el proceso de actualización tradicional no es el más adecuado y, por lo tanto, se requiere de técnicas que permitan una actualización del sistema mientras esté en ejecución.

En este artículo se presentará una técnica para permitir actualizar un sistema en tiempo de ejecución, mediante la carga, descarga y actualización de un conjunto de las clases del sistema. A los sistemas que utilizan esta técnica, se los denominará *Sistemas Dinámicamente Extensibles y Modificables*.

En la sección 2 se presentan algunos trabajos relacionados al tema. En tanto, la sección 3 introduce algunos elementos de discusión utilizados para la implementación del presente trabajo. En la sección 4 se describe la arquitectura propuesta, y el uso de la arquitectura se encuentra en la sección 5. Un ejemplo de la aplicación de esta técnica se presenta en la sección 6. Finalmente, algunas conclusiones están plasmadas en la sección 7.

Algoritmo 1: Solución para la llamada a un método sin conocer su declaración. `InvocarMétodo` es un método virtual de `ClaseDinámica` cuya declaración se conoce en la aplicación y `MiMétodo` es un método de `MiClaseDinámica` cuya declaración no se conoce en la aplicación.

En la Aplicación

```
pObjeto->InvocarMétodo (IDMétodo, param1, param2, ...)
```

En la Unidad de Extensión

```
MiClaseDinámica::InvocarMétodo (IDMétodo, parametros)
    Si (IDMétodo == IDMiMétodo)
       param1 = RecuperarParámetro(parámetros, 1)
       param2 = RecuperarParámetro(parámetros, 2)
       ...
       MiMétodo (parámetro1, parámetro2, ...)
    Fin si
Fin InvocarMétodo
```

2 Trabajos relacionados

La conversión de código fuente[2] a un código ejecutable[3] se realiza de dos formas básicas, o alguna mezcla de éstas: *interpretación* o *compilación*[4].

En la **interpretación**, un programa, llamado intérprete, ejecuta el código fuente analizando línea a línea cada sentencia del programa y convirtiendo la misma a código máquina. De forma pura, esta forma de ejecución agrega una sobrecarga al programa,

[2] un programa escrito en un lenguaje de programación de alto o bajo nivel.

[3] un programa escrito en código máquina.

[4] Java es un ejemplo de un lenguaje de programación que utiliza una mezcla de ambos métodos para la conversión de código fuente en un código ejecutable.

debido a que la conversión a código máquina se realiza al mismo tiempo que la ejecución. De esta manera, cuando se interpreta un lenguaje de alto nivel, la sobrecarga que se agrega se vuelve significativa.

En la **compilación**, la obtención de un programa ejecutable se realiza comúnmente en dos fases separadas: la compilación y el enlace. Durante la compilación, un programa especial, llamado compilador, analiza el programa contenido en el archivo fuente y lo convierte a código objeto, en un archivo especial llamado archivo objeto[5]. Otro programa especial, llamado enlazador, une los archivos objeto y resuelve las direcciones que quedaron pendientes durante la compilación. El resultado de este proceso es un archivo ejecutable que puede ser ejecutado directamente por la computadora [7, 13].

Un intento de agregar dinamismo a una aplicación se basa en el uso de *módulos de carga dinámica*. Para flexibilizar el proceso de enlace, las direcciones de los procedimientos pueden ser definidas en tiempo de ejecución y no durante la fase de enlace. Para permitir esto, el sistema operativo debe ser capaz de cargar archivos objeto (con algún formato especial) y resolver las llamadas a los procedimientos en el código de la aplicación. Este tipo de enlace se conoce como "enlace dinámico" o "enlace perezoso"[6] [7]. Los archivos DLL[7] en Windows y OS/2 [9], y SO[8] en Solaris [4, 7] son ejemplos de módulos de carga dinámica.

El enlace incremental [10, 11] permite enlazar en tiempo de ejecución código objeto que está disponible en tiempo de compilación. Esta propuesta permite administrar de mejor manera los recursos, permitiendo agregar y quitar segmentos de código durante la ejecución, a condición de que los archivos objeto permanezcan inmutables durante este tiempo. Por otra parte, el enlace dinámico permite cambiar los segmentos de código al ser capaz el SO de enlazar la aplicación con código objeto que no estaba disponible en tiempo de compilación.

Algoritmo 2: Declaración de la clase A.

```
class A
{
    public:
       A ()
       { //Inicializa la clase }

       ~A ()
       { // Finaliza la clase }

       unsigned M1 (int parameter1, float parameter2)
       { // Realiza algo }
};
```

[5]Archivo que contiene la versión en código máquina de un archivo fuente
[6]Traducción de *Lazy link*
[7]Acrónimo de *Dynamic Link Library*.
[8]Acrónimo de *Shared Objects*.

El lenguaje C++ no posee un soporte directo para la carga dinámica de módulos, aunque se han realizado varios intentos para poder darle esta característica. Sin embargo, como menciona Hjàlmtỳsson [6], "los enlazadores dinámicos actuales rompen la seguridad de tipos de C++, dado que están orientados hacia funciones en vez de clases".

El trabajo presentado por Doward *et al.* [3] representa una extensión al enlace dinámico, que trabaja a nivel de clases, y preserva la seguridad de tipos de C++. Su solución permite que una nueva clase derivada de una clase base conocida pueda ser cargada dinámicamente en un programa. Gracias a que todas las llamadas a los métodos de la clase base son verificadas estáticamente, la seguridad de tipos se preserva en cualquier llamada a un método de una clase derivada. Una limitación de la solución propuesta es la imposibilidad de reemplazar una clase cargada anteriormente con una nueva versión.

Objetos Dinámicos en Tiempo de Ejecución (ODTE) es una solución propuesta por Ed Smetak y Jean Caputo de Nanosoft Corporation [12] para el desarrollo de aplicaciones MFC que puedan cargar y descargar dinámicamente clases en tiempo de ejecución. Todas las clases dinámicas[9] heredan de una misma clase y son manejadas por dos clases estáticas[10] de la aplicación, una orientada al manejo de clases dinámicas y otra orientada al manejo de los objetos instanciados por estas clases. Por otro lado, con ODTE es posible cargar clases que no tengan una declaración conocida en tiempo de compilación, pero sólo se permite utilizar un número reducido de métodos estándares. Pero, si la declaración de la clase es conocida en tiempo de compilación, es posible acceder a todos los métodos de ésta. Gìsli Hjàlmtỳsson y Robert Gray de los laboratorios de AT&T [6] presentan Clases C++ Dinámicas, una técnica que extiende el trabajo de Doward *et al.* [3] para permitir reemplazar una clase dinámica cargada anteriormente y poder mantener varias versiones de la misma en el programa. Cada clase dinámica posee dos partes: una clase abstracta que funciona como interfaz y una o más clases hijas de la clase abstracta que implementan la interfaz. El núcleo de la implementación es un template genérico que sirve como *clase-proxy* o "puntero inteligente" a cada clase dinámica (interfaz e implementación). La clase-proxy es la que carga la implementación de las clases en el espacio de direcciones del programa, de forma que las llamadas a las operaciones de la clase interfaz sean correctamente redirigidas hacia la implementación adecuada.

La carga dinámica de clases en tiempo de ejecución permite modificar la jerarquía de clases de la aplicación. Para llamar a un método de una clase es necesario conocer su declaración en tiempo de compilación, dado que en C++ la verificación de tipos la realiza el compilador. Por este motivo, los trabajos presentados en el área [12, 6, 5, 3] han utilizado el polimorfismo como el motor central para la llamada a métodos de clases dinámicas. De esta forma, la aplicación siempre conoce la declaración de la clase (o por lo menos una interfaz) y existen elementos encargados de enlazar la implementación con la declaración.

[9] Clase que puede ser cargada/descargada dinámicamente en tiempo de ejecución.
[10] Clases que están enlazadas al programa en tiempo de compilación.

Algoritmo 3: Implementación del método `M2` de la clase `B`.

```
void B::M2
{
    // ...
    A* pA = new A ();
    // ...
    unsigned respuesta = pA->M1 (5, 2.6);
    // ...
    delete pA;
    // ...
}
```

Una consecuencia directa de la anterior forma de implementación es la necesidad de conocer la declaración o interfaz de las clases en tiempo de compilación, aun cuando no se conozca su implementación. Cuando se instancia un nuevo objeto, en realidad se instancia un objeto de una clase derivada. Como menciona Hjàlmtỳsson [6], "cada nueva implementación o actualiza una clase existente (una nueva versión) o introduce una nueva clase (un nuevo tipo) que utiliza una interfaz conocida".

Gracias al uso de clases dinámicas, es posible implementar aplicaciones cuyas clases pueden ser modificadas en tiempo de ejecución. Sin embargo, un problema que presentan las técnicas anteriormente propuestas es la obligatoriedad de conocer una interfaz a estas clases, la cual no puede ser modificada. Si se considera el caso de una aplicación que permite el uso de clases dinámicas dentro de "scripts", las anteriores técnicas restringen al usuario a utilizar los métodos declarados en las interfaces, de las clases dinámicas, que fueron utilizadas en el momento de compilar la aplicación.

Como una extensión a los anteriores trabajos, la técnica propuesta en este artículo utiliza la noción de clases dinámicas, pero que están empaquetadas en *Unidades de Extensión*. Para una aplicación, cada unidad de extensión podrá ser vista como un componente de software, dado que ésta empaqueta toda la información necesaria para ser enlazada a una aplicación. Una ventaja importante del uso de unidades de extensión es la posibilidad de utilizar clases dinámicas aun cuando la declaración no sea conocida por la aplicación, evitando así la necesidad de interfaces.

3 Unidades de extensión

En esta sección se introduce el concepto de *unidad de extensión* como "un conjunto de clases que, tratadas como una unidad, pueden ser desarrolladas de forma independiente y enlazarse a cualquier SDEM en tiempo de ejecución". Dentro de cada unidad de extensión debe existir una única clase dinámica; es decir, una clase que será dinámicamente incluida en la jerarquía de clases de la aplicación. La aplicación sólo administrará clases dinámicas e instancias de estas clases, mientras que las otras clases de la unidad de extensión serán administradas por la clase dinámica (ver Figura 1).

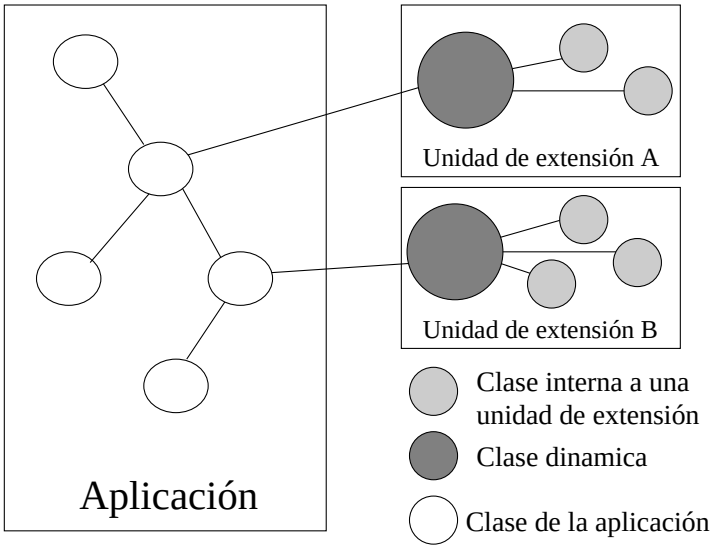

Figura 1: Diagrama de la relación entre una aplicación y una unidad de extensión. Como es posible observar, cada unidad de extensión posee una única clase dinámica, la cual puede estar relacionada con otras clases propias de la unidad de extensión pero que no pueden ser accedidas desde fuera de esta, mientras las clases dinámicas pueden ser accedidas desde la aplicación (u otra unidad de extensión).

3.1 Llamada a métodos de clases dinámicas

Toda clase dinámica debe heredar de una misma clase (una especie de clase "proxy"), la que declara un conjunto mínimo de métodos virtuales que la aplicación utiliza para interactuar con cualquier clase dinámica. Por polimorfismo, una solicitud de la aplicación puede ser derivada hasta una clase dinámica de una unidad de extensión. Sin embargo, si la clase dinámica contiene métodos que no están declarados en la clase proxy, no es posible llamarlos de la misma forma. Para solucionar el anterior problema se requiere un mecanismo que permita llamar a un método sin necesidad de conocer su declaración.

La posibilidad de tener métodos con un número variable de argumentos, facilita la llamada a un método pasándole cualquier parámetro sin que el compilador realice una verificación de la llamada. Esta herramienta es un arma de doble filo, porque compromete seriamente la seguridad de tipos de C++, pero facilita el trabajo. Uniendo esta técnica con la descrita anteriormente, es posible declarar en la clase proxy un método virtual con número variable de argumentos, que será utilizado para realizar llamadas a otros métodos (ver Algoritmo 1), pasando como parámetro un identificador al método en la clase dinámica al que se desea llamar. Una vez dentro de la clase dinámica, es posible construir una llamada al método solicitado recuperando los parámetros con `va_arg`, debido a que en ese contexto ya se conoce la declaración del método solicitado.

Un último problema a resolver es el manejo de los tipos de retorno. Para solucionar

esto se utiliza una estructura `union` con todos los tipos básicos del lenguaje. En la aplicación se recupera el tipo apropiado y, en el caso de punteros a objetos, se realiza un modelado al tipo apropiado. Para una idea más clara ver el Algoritmo 6, donde una llamada al método `CA_M1` se convierte a un entero sin signo mediante `ToUInt()` y más abajo a un valor lógico mediante `ToUInt()`, ambos métodos obtienen el campo adecuado de la estructura `union` contenida en la clase `CMultivalueObject`.

Algoritmo 4 : Implementación de la clase dinámica `ClaseDinamicaA`, la cual hereda de `CDynamicClass`. Se puede observar que la interacción de la aplicación con una clase dinámica se realiza mediante el método virtual `CallMethod`, el cual pertenece a `CDynamicClass`, y que se encarga de realizar la llamada al método deseado. Este método recibe como entradas un puntero al objeto sobre el cual el usuario desea realizar una acción, un identificador del método a llamar y una lista de los parámetros del método. El constructor de la clase debe declarar todos los métodos que de la clase dinámica que pueden ser accedidos desde fuera de la unidad de extensión a la que pertenece.

```
#define CA_M1 1000001
ClaseDinamicaA::ClaseDinamicaA (CClassInfo* theClassInfo):  CDynamicClass (theClassInfo)
{
     CMethodSpecification* pMethod;
     pMethod = RegisterMethod (CA_M1, ‘‘M1’’, ‘‘Un método’’, CMethodSpecification::cUINT, 2);
     pMethod->AddParameter (CMethodSpecification::cInt, ‘‘parameter_1’’, 1, ‘‘int parameter1’’);
     pMethod->AddParameter (CMethodSpecification::cFloat, ‘‘parameter_2’’, 2, ‘‘float parameter2’’);
}

CMultivalueObject ClaseDinamicaA::CallMethod(void* *pObject, unsigned iMethodID,
         va_list pListOfParameters)
{
     CMultivalueObject result;
     InitCriticalSection ();
     if (iMethodID == CA_M1)
     {
         int parameter1 = va_arg (pListOfParameters, int);
         float parameter2 = va_arg (pListOfParameters, float);
         result = CMultivalueObject (((A*)pObject)->M1(parameter1, parameter2));
     }
     FinishCriticalSection ();
     return result;
}

void* ClaseDinamicaA::GetNewObject()
{
     return new A ();
}

void ClaseDinamicaA::FinishObject(void *theObject)
{
     delete (A*) theObject;
}
```

3.2 Manejo de las versiones

Al actualizar una unidad de extensión es importante definir qué acciones se deben tomar con los objetos de la clase dinámica que será actualizada. Según Hjàlmtỳsson [6] existen tres enfoques para este problema. Un primer enfoque *(a)* es sólo habilitar la actualización de una clase cuando ya no existan objetos de esta clase. Otro enfoque

(b) consiste en actualizar los objetos de la antigua clase y convertirlos a objetos de la nueva clase. El último enfoque *(c)* simplemente no toma ninguna acción, es decir que mantiene la antigua clase en memoria junto con la nueva clase, pero cada vez que se cree un nuevo objeto se utilizará la nueva clase.

Algoritmo 5 : Implementación de los métodos `GetClassID` que devuelve el código identificador de la clase, `GetDescription` que retorna una cadena descriptiva de la clase, `Init` que crea un objeto de la clase dinámica y retorna su dirección, y `Finish` que destruye el objeto de la clase dinámica.

```
#include ''ClaseDinamicaA.h''
ClaseDinamicaA *pClaseDinamica = NULL;
extern ''C'' int APIENTRY
        DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID lpReserved)
{
    return 1; // ok
}
extern ''C'' void* WINAPI Init (void* classInfoP)
{
    pClaseDinamica = new ClaseDinamicaA ((CClassInfo*)classInfoP);
    return pClaseDinamica;
}
extern ''C'' char* WINAPI GetDescription()
{
    return ''Clase A'';
}
extern ''C'' CLSID WINAPI GetClassID()
{
    /*
    Vendor ID: 34E300C1 Oscar Antezana Chavez
    Class Type:  FFEE Clases de Prueba
    Class ID: 1001 CA
    Version:  01 Primera versión pública
    Type:  01 Primera versión interna
    Key Value:  00000000 Sin compatibilidad
    Passwd:  0000 Sin clave
    */
    // ID: 34E300C1-FFEE-1001-0101-00000000000
    CLSID clsidID =  0x34e300c1, 0xFFEE, 0x1001,
         0x01, 0x01, 0x0, 0x0, 0x0, 0x10, 0x0, 0x0  ;
    return clsidID;
}

extern ''C'' void WINAPI Finish ()
{
    delete pClaseDinamica;
}
```

La propuesta *(a)* no es la más adecuada, debido a que se restringe el momento en que se puede realizar una actualización; mientras que la propuesta *(c)* puede permitir demasiadas versiones de una misma clase en memoria, lo cual representa un gasto inútil de recursos. Por estos motivos, se ha decidido utilizar la propuesta *(b)*, es decir convertir los objetos de la antigua versión a la nueva. Para realizar esto se almacena en un buffer temporal toda la información necesaria del objeto para crear una nueva versión del mismo que posea toda la información del objeto original. De esta forma, la interacción con los nuevos objetos se realizará de forma transparente, puesto que la nueva versión del objeto sabe cómo responder a las solicitudes de los métodos de la antigua clase, así como también a los de la nueva clase.

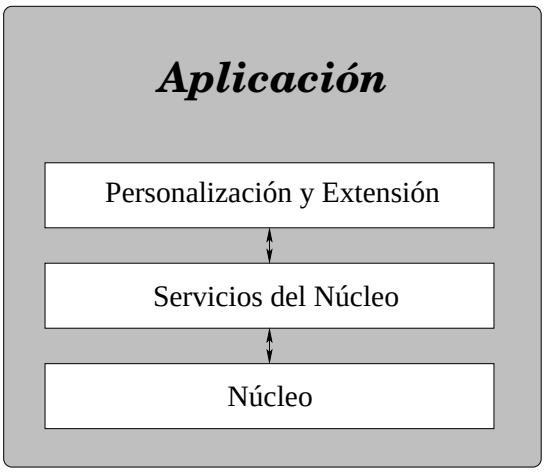

Figura 2: Diagrama de la arquitectura SDEM, donde se pueden observar la interacción entre los tres subsistemas de la arquitectura: Núcleo, Servicios del Núcleo y Personalización y extensión.

4 La arquitectura SDEM

La técnica presentada en este artículo está centrada en una arquitectura base que puede ser utilizada por otras aplicaciones para desarrollar sistemas dinámicamente extensibles y modificables. Un *sistema modificable* se definirá como "aquel sistema que brinda al usuario herramientas para adaptar y adecuar un sistema de software a sus necesidades particulares". Un *sistema extensible* será "un sistema que provee herramientas orientadas al usuario para la administración de unidades de extensión". Cuando un sistema es extensible y modificable en tiempo de ejecución, se lo denominará *Sistema Dinámicamente Extensible y Modificable* (SDEM).

La arquitectura SDEM propuesta consta de tres (3) subsistemas[11] (ver Figura 2), de los cuales el más importante es el subsistema núcleo, mientras que los otros dos subsistemas permiten interactuar con el núcleo a diferentes niveles, ya sea desde código, o desde la interfaz de usuario de la aplicación. El incluir elementos de interacción con el núcleo representa un mecanismo de seguridad para evitar un mal uso de la arquitectura, como por ejemplo intentar llamar a un método de un objeto mientras éste se está actualizando; pero también agrega un nivel de trabajo extra, produciendo como consecuencia una mayor demora en la respuesta de las operaciones efectuadas por el núcleo.

A continuación se realiza un análisis de cada uno de los subsistemas que forman parte de la arquitectura SDEM.

[11]Existe un cuarto subsistema llamado *Clases de soporte.*, que no será considerado parte de la arquitectura, dado que sólo contiene clases de utilidades como manejo de listas, manejo de hilos y otros similares que no son relevantes en la técnica presentada en este artículo.

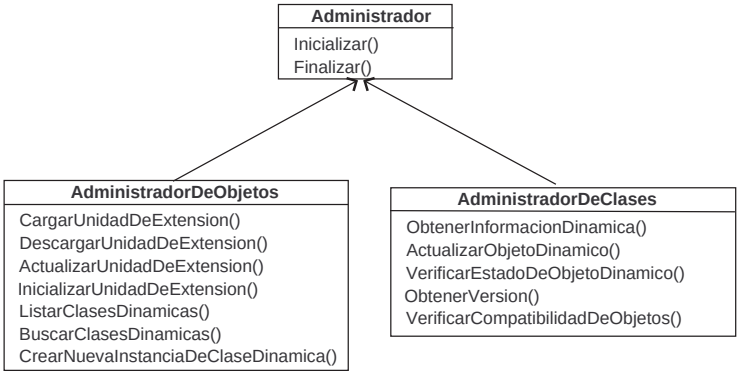

Figura 3: Diagrama de clases mostrando los métodos más importantes de las clases `AdministradorDeClases` y `AdministradorDeObjetos`. Estas clases forman parte del subsistema Núcleo.

4.1 Núcleo

Este subsistema es el encargado de realizar el manejo de las clases y objetos dinámicos. Para este fin, el Núcleo utiliza dos clases: `AdministradorDeClases` y `AdministradorDeObjetos` (ver Figura 3). El **Administrador de clases** se encarga de manejar la carga, descarga y actualización de clases; mientras que el **Administrador de objetos** permite crear, destruir y actualizar los objetos de las clases dinámicas. Para poder manejar los objetos y clases dinámicas es necesario mantener un registro de cada uno de ellos. Por cada clase dinámica cargada en el sistema existe un objeto `InformaciónDeClase` que permite obtener información de la clase, así como manejarla (descargarla, actualizarla, etc.). De forma similar, por cada objeto de una clase dinámica existe un objeto de la clase `InformaciónDeObjeto` el cual permite manejar la referencia al mismo; es decir, conoce dónde está el objeto asociado.

El Núcleo es el que provee un completo control sobre las unidades de extensión y las clases dinámicas, así como una amplia información de las mismas. Debido a que se maneja un registro de cada método de una clase dinámica, es posible conocer, en tiempo de ejecución, los parámetros, tipos de los parámetros, tipos de retorno de cada método de cada clase dinámica. Esta información está contenida en cada clase dinámica, la que debe obligatoriamente registrar los métodos que pueden ser invocados desde la Aplicación.

4.2 Servicios del Núcleo

Cuando la Aplicación desea interactuar con el Núcleo, envía sus solicitudes al subsistema Servicios del Núcleo, quien se encarga de recoger esta solicitudes y enviarlas al administrador de objetos, al administrador de clases, a una clases dinámica o a un objeto de estas clases. Las solicitudes recibidas son tratadas como transacciones; es decir, como una unidad indivisible. Esta forma de trabajo asegura una interacción confiable

entre la Aplicación y el Núcleo.

Es importante destacar que la existencia de este subsistema está dada sólo por propósitos de seguridad y mantenimiento de la integridad de la arquitectura. Por otra parte, debido a que la existencia de este subsistema no es esencial para el correcto funcionamiento de un SDEM, dado que todo el trabajo se realiza en el subsistema Núcleo y que además se utiliza una arquitectura de capas donde el subsistema Núcleo trabaja sin conocer de la existencia del subsistema Servicios del Núcleo, este subsistema se ha implementado como una clase dinámica. De esta forma el usuario puede reemplazar el subsistema Servicios del Núcleo por una implementación más adecuada.

Existe una cola de solicitudes recibidas y un monitor que va verificando la cola constantemente en busca de solicitudes que puedan ser procesadas por el Núcleo. Cada vez que existe una solicitud preparada para ser procesada, se la traslada a una lista de solicitudes en procesamiento y se pasa la misma al Núcleo. Una vez procesada, la transacción es trasladada a una lista de transacciones procesadas.

Aunque el proceso anterior puede ser demasiado pesado, un manejo de hilos de ejecución y semáforos permite que el manejo de las listas sea realizado sólo cuando sea necesario y el hilo encargado de este trabajo se mantenga suspendido hasta que sea necesario ejecutarlo, *i.e.* cuando se reciba una transacción del usuario.

4.3 Personalización y Extensión

Este último subsistema provee un conjunto de diálogos para poder interactuar a nivel de interfaz de usuario con el subsistema Servicios del Núcleo. Los componentes de este subsistema incluyen diálogos para manejar las unidades de extensión (cargar, descargar, actualizar y ver información) y monitorear las listas de transacciones (ver información y estado de cualquier transacción).

Al igual que el subsistema Servicios del Núcleo, este subsistema ha sido implementado como una clase dinámica.

5 Interacción con la arquitectura

Para poder interactuar con la arquitectura propuesta, se han desarrollado varios archivos DLL (uno para cada subsistema) que deben ser incluidos en la Aplicación que utilizará la arquitectura. Dos son las clases más importantes que son utilizadas por el programador: `CSDEMBaseApp` que debe ser utilizada como la clase padre de la clase Aplicación, y `CSDEM` que provee el método `CreateSynchronousTransaction`, el cual es encargado de llevar una transacción hasta el subsistema Servicios del Núcleo y retornar la respuesta cuando la transacción hubiese sido completada.

Cada posible transacción posee un identificador; por ejemplo, para instanciar un objeto de una clase se debe crear una transacción del tipo: `SDEMTO_CREATEOBJECT`. Existe una gran variedad de transacciones, desde cargar una unidad de extensión hasta consultar el número de métodos que una clase dinámica posee.

Algoritmo 6: Implementación del método `M2` de la clase `B` utilizando clases dinámicas.

```
#define CA_M1 1000001
void B::M2
{
     // ...
     CLSID clsid =  0x00000000, 0xFFEE, 0x1001,
          0x01, 0x00, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0  ;
     CLSID* pCA =
         (CLSID*)pSDEM->CreateTransaction
         (SDEMTE_FINDCOMPATIBLECLASS, &clsid).ToPointer ();
     CLSID* pA =
         (CLSID*)pSDEM->CreateTransaction (SDEMTE_CREATEOBJECT, pCA).ToPointer ();
     // ...
     unsigned respuesta =
         pSDEM->CreateTransaction(SDEMTE_CALLMETHOD,
             pA, CA_M1, 5, 2.6).ToUInt();
     // ...
     pSDEM->CreateTransaction (SDEMTE_DESTROYOBJECT, pA).ToBool ();
     // ...
}
```

6 Ejemplo de una unidad de extensión

En este apartado se mostrará un ejemplo de la forma en que una unidad de extensión debe ser implementada. El ejemplo presentado consistirá de una clase `A` que posee un método llamado `M1`, como se muestra en el algoritmo 2, y una clase `B` con un método llamado `M2` (ver Algoritmo 3).

Si se desea implementar la clase `A` como una unidad de extensión, de forma que pueda realizar la misma actividad descrita en `M2`, el desarrollador debe realizar un conjunto de pasos que comienzan por llevar la clase `A` (declaración e implementación) a una librería de enlace dinámico (DLL). Una vez hecho ello, se debe crear una clase que herede de `CDynamicClass` que permita interactuar con la clase `A`. En este último paso, las actividades a realizar son: asociar al método `M1` un identificador numérico único en el contexto de la clase (en este caso se asoció a `1000001`), registrar este método en el constructor de la clase dinámica y sobrecargar los métodos: `CallMethod`, `Init` y `Finish`, mas aquellos necesarios para las funcionalidades deseadas (ver Algoritmo 4).

Para completar la unidad de extensión se debe implementar cuatro funciones públicas de la DLL (`GetClassID`, `GetDescription`, `Init` y `Finish`). Por lo general, esto se realiza en el mismo lugar donde se implementó la función `DLLMain`. El código respectivo se encuentra en Algoritmo 5.

Finalmente, la implementación del método `M2` de la clase `B` utilizando `A` como una unidad de extensión está en el Algoritmo 6.

Pese a que el anterior trabajo es realizado de forma manual por el desarrollador, es posible crear un programa que realice de forma automática todos los pasos descritos anteriormente.

En [1] se presenta el análisis, diseño e implementación de un sistema para el monitoreo de dispositivos USB que fue desarrollado utilizando la arquitectura presentada en este artículo.

7 Conclusiones

Existen diversas técnicas para cargar/descargar clases de forma dinámica en una aplicación C++ [6, 12, 3]. Cada una de estas técnicas presentan ventajas y desventajas con respecto a la arquitectura SDEM propuesta. Una de las limitantes más importantes de las técnicas anteriores es la necesidad de conocer una interfaz de las clases dinámicas en tiempo de compilación. El uso de una interfaz en tiempo de compilación permiten mantener la seguridad de tipos y obtener un rendimiento adecuado contra la imposibilidad de llamar a métodos que pueden estar implementados en la clase dinámica, pero no declarados en la clase proxy.

Otra limitante de las anteriores técnicas es la falta de control sobre las clases dinámicas. Salvo en ODTE [12] las otras técnicas no proveen clases encargadas de monitorear las clases dinámicas y los objetos de éstas, aún así, las clases encargadas de este trabajo en ODTE son accedidas directamente por el desarrollador, quien puede corromperlas con facilidad. El presente trabajo propone una arquitectura mucho más elaborada que las anteriores, con tres subsistemas que permiten interactuar a diferentes niveles con las clases dinámicas y sus objetos.

Aunque la arquitectura presentada da soporte a todos los elementos necesarios para desarrollar SDEMs, sacrifica dos importantes elementos como son: la seguridad de tipos (que se pierde al utilizar funciones con número variable de argumentos) y la facilidad de desarrollo desde el punto de vista del desarrollador.

Pese a las limitaciones del trabajo, que podrían ser base de estudio para futuras investigaciones, su uso es adecuado para aplicaciones donde el número de clases dinámicas es reducido, por ejemplo, donde las unidades de extensión representan subsistemas o módulos completos. Por otro lado, provee soporte para el desarrollo de unidades de extensión y facilita el desarrollo de aplicaciones que las utilicen, gracias a que con cada unidad de extensión se incluye amplia información sobre la clase dinámica.

Referencias

[1] O. Antezana. Sistemas dinámicamente extensibles y modificables. Tesis de Licenciatura, Universidad Católica Boliviana, 2000.

[2] G. Booch. *Object Oriented Analysis and Design - with Applications.* Benjamin - Cummings, segunda edición, 1994.

[3] S. M. Doward, R. Sethi, y J. E. Shopiro. Adding new code to a running C++ program. En *Proceedings of the USENIX C++ Conference*, 1990.

[4] R.A. Gingell, M. Lee, X.T. Dang, y M.S. Weeks. Shared libraries in SunOS. En *Proceedings of the USENIX Summer Conference.*, 1987.

[5] T.C. Goldstein y A. D. Sloane. The object binary interface - C++ objects for evolvable shared class libraries. Reporte Técnico TR-94-26, Sun Microsystems Laboratories, Mountain View, California, 1994.

[6] G. Hjàlmtỳsson y R. Gray. Dynamic C++ classes. a lightweight mechanism to update code in a running program. Reporte técnico, AT&T Labs, 1997.

[7] J. Kempf y P.B. Kessler. Cross-address space dynamic linking. Reporte Técnico TR-92-2, Sun Microsystems Laboratories., Mountain View, California., 1992.

[8] J. Lang y D. B. Stewart. A study of the applicability of existing exception-handling techniques to component-based real-time systems. *ACM Transactions on Programming Language and Systems*, 20(2), 1998.

[9] G. Letwin. Dynamic linking in OS/2. *Byte*, 13(4), Abril, 1998.

[10] J.J. Putterss y H.H. Goguen. Incremental loading of subroutines at runtime. Reporte técnico, AT&T Bell Laboratories, 1986.

[11] R.W. Quong. The design and implementation of an incremental linker. Reporte Técnico CSL-TR-88-381, Computer Systems Laboratory. Stanford University, 1989.

[12] E. Smetak y J. Caputo. Using dynamic objects to build extensible MFC applications. *Microsoft Systems Journal*, Julio, 1997.

[13] W. Wilson y R.A. Olsson. An approach to genuine dynamic linking. *Software Practice and Experience*, 21(4), Abril, 1991.