Implantación del Kernel de OpenMusic bajo Linux

GERARDO MAURICIO SARRIA
JOSE FERNANDO DIAGO

PONTIFICIA UNIVERSIDAD JAVERIANA
FACULTAD DE INGENIERIA
INGENIERIA DE SISTEMAS Y COMPUTACION
SANTIAGO DE CALI
2001

Implantación del Kernel de OpenMusic bajo Linux

GERARDO MAURICIO SARRIA
JOSE FERNANDO DIAGO

Tesis de grado para optar al título de
Ingeniero de Sistemas y Computación

Director
CAMILO RUEDA
Ingeniero de Sistemas y Computación

PONTIFICIA UNIVERSIDAD JAVERIANA
FACULTAD DE INGENIERIA
INGENIERIA DE SISTEMAS Y COMPUTACION
SANTIAGO DE CALI
2001

Santiago de Cali, Junio 15 de 2001

Ingeniero
ANDRÉS JARAMILLO BOTERO
Decano Académico de la Facultad de Ingeniería
Pontificia Universidad Javeriana
Ciudad

Certifico que el presente proyecto de grado, titulado "Implantación del Kernel de OpenMusic bajo Linux" realizado por GERARDO MAURICIO SARRIA y JOSE FERNANDO DIAGO, estudiantes de Ingeniería de Sistemas y Computación, se encuentra terminado y puede ser presentado para sustentación.

Atentamente,

Ing. CAMILO RUEDA
Director del Proyecto

Santiago de Cali, Junio 15 de 2001

Ingeniero
ANDRÉS JARAMILLO BOTERO
Decano Académico de la Facultad de Ingeniería
Pontificia Universidad Javeriana
Ciudad

Por medio de ésta, presentamos a usted el proyecto de grado titulado "Implantación del Kernel de OpenMusic bajo Linux" para optar al título de Ingeniero de Sistemas y Computación.

Esperamos que este proyecto reúna todos los requisitos académicos y cumpla el propósito para el cual fue creado, y sirva de apoyo para futuros proyectos en la Universidad Javeriana relacionados con la materia.

Atentamente,

GERARDO MAURICIO SARRIA JOSE FERNANDO DIAGO

ARTICULO 23 de la Resolución No 13 del 6 de Julio de 1946
del Reglamento de la Pontificia Universidad Javeriana.

"La Universidad no se hace responsable por los conceptos emitidos por sus alumnos en sus trabajos de Tesis. Sólo velará porque no se publique nada contrario al dogma y a la moral Católica y porque las Tesis no contengan ataques o polémicas puramente personales; antes bien, se vea en ellas el anhelo de buscar la Verdad y la Justicia"

Nota de Aceptación:

Aprobado por el comité de Trabajo de Grado en cumplimiento de los requisitos exigidos por la Pontificia Universidad Javeriana para optar al título de Ingeniero de Sistemas y Computación.

ANDRÉS JARAMILLO BOTERO
Decano Académico de la Facultad de Ingeniería

CAMILO RUEDA
Director de la Carrera de Ingeniería de Sistemas y Computación

CAMILO RUEDA
Director de Tesis

GABRIEL TAMURA
Jurado

ANTAL BUSS
Jurado

Gerardo Mauricio Sarria

A Jose y a mí, por seguir juntos desde Preescolar.

Jose Fernando Diago

A Gerardo, Llanos, Vacca, Javier y Carlitos, por mostrarme lo que puedo ser.

Agradecimientos

Los autores expresan sus agradecimientos:

- A Camilo Rueda, profesor y director de la carrera de Ingeniería de Sistemas y Computación de la Universidad Javeriana, director del proyecto, por su valioso apoyo y conocimiento brindado durante la realización del trabajo.
- A Carlos Agon, uno de los creadores de OpenMusic, por su ayuda incondicional desde Francia.
- A Antal Buss, profesor de la carrera de Ingeniería de Sistemas y Computación de la Universidad Javeriana, por sus ideas y consejos durante la implantación del código.
- A Espen S. Johnsen, creador de CLG, por sus completas explicaciones acerca del manejo de la librería Gtk+ en Lisp.
- A Pierre R. Mai, desarrollador de CMUCL, por su orientación en momentos críticos.
- Al Grupo Avispa.
- A todas aquellas personas que de una u otra forma colaboraron en la realización del presente trabajo.

Índice general

Índice de figuras

Índice de Anexos

Resumen

En este documento se pretende mostrar los detalles de la Implantación del Kernel de OpenMusic bajo Linux, siguiendo una metodología propuesta por los autores; la implantación en este caso hace referencia al porte de una aplicación originalmente desarrollada en MacOS.

Se da inicio al porte familiarizándose con la aplicación en MacOS, como lo haría cualquier otro usuario pero prestando particular atención en los detalles que más adelante serán decisivos en la implementación.

A continuación se explora el código y se procede a escoger las herramientas de desarrollo. Las herramientas de desarrollo escogidas son CMUCL, un compilador de Lisp para Unix que ha sido desarrollado durante más de una década en el Departamento de Ciencias de la Computación de la Universidad Carnegie Mellon, y GTK+ para la interfaz gráfica de la aplicación.

Contando con las herramientas de desarrollo adecuadas se diseñaron e implementaron los módulos de la aplicación (tanto en su parte gráfica como en su parte operativa) en Linux y se llevaron a cabo las pruebas pertinentes para constatar que el porte de la aplicación presenta las mismas características y funcionalidad que la aplicación original.

Introducción

La Implantación del Kernel de OpenMusic bajo Linux hace referencia en este proyecto al porte de la aplicación de la plataforma original (MacOS) a una nueva plataforma (Linux).

Al Portar un programa de una plataforma informática a otra se pretende proveer a los usuarios de la plataforma destino con una herramienta lo más parecida posible, en aspecto y funcionalidad, a la herramienta original.

Dependiendo del lenguaje en el que se encuentre escrito el programa original, el compilador usado, la portabilidad que ofrezcan el lenguaje y el compilador mismo, las librerías usadas y el uso de recursos dependientes de la plataforma, suele ser necesario en algunos casos una casi completa re-escritura del código del programa. En el caso del porte de Open Music se hizo necesario reescribir una buena parte del código, tratando siempre de preservar y respetar la estructura original del programa.

La plataforma de origen involucrada (MacOS) provee al programador de muchos recursos suficientemente poderosos y bastante simples de usar pero totalmente dependientes de la plataforma misma, hecho que suele ser motivo de una reimplementación completa de estos recursos en la plataforma destino (Linux x86). Linux como plataforma ofrece recursos igualmente poderosos y bastante simples de usar y que además suelen ser portables en una amplia gama de plataformas que suele incluir casi cualquier versión de Unix y Windows, lo cual en su mayor parte se debe a la disponibilidad del código fuente de los programas para Linux junto con las herramientas para compilarlos en distintas plataformas.

En el caso particular del porte de OpenMusic se escogió un compilador disponible para una gran variedad de versiones de Unix y una librería para interfaces gráficas que puede ser usada tanto en Unix como en Windows, lo anterior hace posible que en

un futuro OpenMusic pueda ser portado a otras plataformas permitiendo extender su uso a un mayor número de usuarios.

El proyecto fue desarrollado a partir de una metodología para portes propuesta por los autores y presentada como un anexo a este trabajo; el alcance del proyecto no incluía el desarrollo de una metodología, pero esta labor fue justificada por la necesidad de una guía para el desarrollo del proyecto y para su uso en futuros trabajos.

1 Consideraciones Generales

En este capítulo se describen brevemente el lenguaje de programación en el que se desarrollo OpenMusic y los Sistemas Operativos involucrados en el presente Trabajo de Grado.

1.1. Common Lisp

Lisp (*the LISt Processing language*) es un lenguaje de programación del paradigma funcional creado por John McCarthy en 1958. Fue originalmente desarrollado como un lenguaje interpretado por lo que un interpretador debe estar presente en el momento en que se introducen los datos para que el computador inmediatamente procese e imprima una respuesta.

Un programa en Lisp puede ser tratado como un dato y puede ser usado como entrada a otros programas o procedimientos directamente. Y debido a la sintaxis poco restringida de Lisp, los programas son fácilmente entendibles para el usuario.

La expresividad de Lisp está caracterizada por un poderoso conjunto de tipos incorporados (listas, cadenas, diccionarios y una gran cantidad de tipos numéricos), cadenas de documentación (Lisp permite al programador poner una cadena al principio de una función con el propósito de documentarla), manejo de excepciones, múltiples espacios para nombres (debido al uso de paquetes) y programación orientada a objetos (usando CLOS: *Common Lisp Object System*).

Common Lisp Object System (CLOS) es una extensión orientada a objetos de Common Lisp basada en funciones genéricas, herencia múltiple, declaración de métodos y protocolo de meta-objetos.

Entre los conceptos más importantes de CLOS (y destacados dentro de este proyecto) se encuentran:

1.1.1. Clase

Una *clase* es un tipo de dato que define las características y comportamiento de un objeto.

Cada clase tiene una lista de atributos, los cuales son objetos que se usan para guardar información relevante al objeto que define la clase.

Las clases pueden heredar las características y comportamiento de una o varias clases. Para este caso, la clase que recibe herencia es denominada *subclase*, mientras que la clase que da la herencia es llamada *superclase*.

En Common Lisp, las clases tiene la siguiente notación:

```
(defclass <nombre> (<lista de superclases>)
   (<lista de atributos>)
   (<opciones de la clase>))
```

Para efectos de comprensión de las clases en algunos problemas, durante el presente documento se usa la representación visual de Rumbaugh [RBP+91]. Luego las clases, gráficamente, se ven como se muestra en la figura 1.1.

1.1.2. Instancia

Los objetos que tienen las características y comportamiento de un clase se les llama *instancias* de la clase.

Para crear la instancia de una clase en Common Lisp, se usa la función:

```
(make-instance <nombre de la clase>
               <valores iniciales de los atributos>)
```

Gráficamente (según la notación de [RBP+91]) las instancias se ven según la figura 1.2.

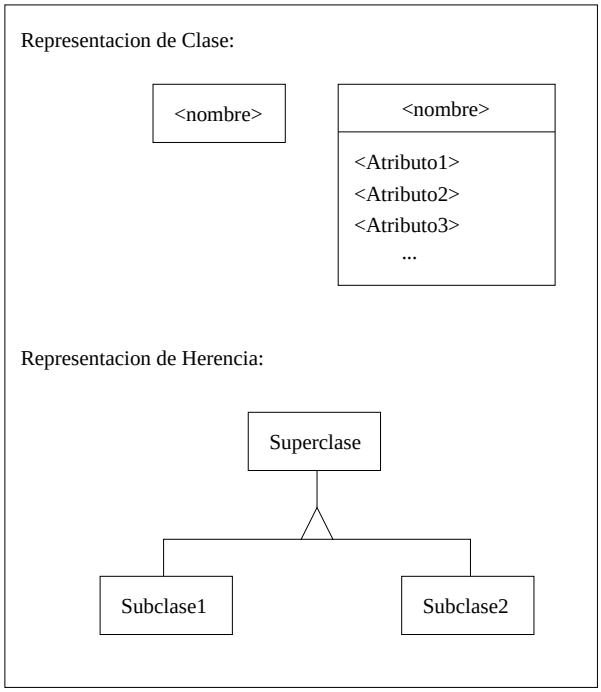

Figura 1.1: Representación de Clases y Herencia.

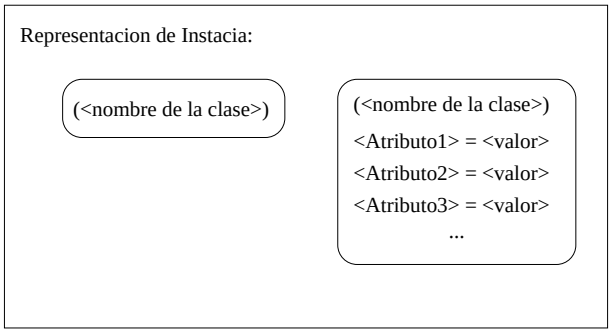

Figura 1.2: Representación de Instancias.

1.1.3. Función Genérica

Una *función genérica* es una función que ejecuta procedimientos u operaciones según el objeto argumento que le sea dado.

Las funciones genéricas se construyen con la macro:

```
(defgeneric <nombre> (<lista de argumentos>)
   <opciones o descripción del método>)
```

1.1.4. Método

Los *métodos* definen las operaciones y procedimientos de la función genérica.

La teoría de programación orientada a objetos tiene como una de sus características principales el polimorfismo. Cuando un mismo método va a ser invocado por distintos objetos, se debe crear una función genérica para este método.

Los métodos se definen así:

```
(defmethod <nombre> <tiempo en que se ejecuta> (<lista de argumentos>)
   <ducumentación>
   <cuerpo del método>)
```

El `<tiempo en que se ejecuta>` es un modificador opcional que puede ser `:before`, `:around` o `:after`, el cual dice cuando se va a ejecutar el método.

A diferencia de otros lenguajes de programación orientados a objetos, en CLOS los métodos no se definen dentro de la clase.

1.1.5. Metaclase

Un objeto es siempre instancia de una clase. A su vez, una clase puede verse como un objeto que es instancia de otra clase: su *metaclase*.

Cuando se va a definir la metaclase de una clase, se coloca en las `<opciones de la clase>` la línea

```
(:metaclass <nombre de la metaclase>)
```

De forma gráfica, las metaclases se muestran como en la figura 1.3 (adaptado de [RBP+91]).

En la figura 1.3, `clase1` es la metaclase de la `clase2`.

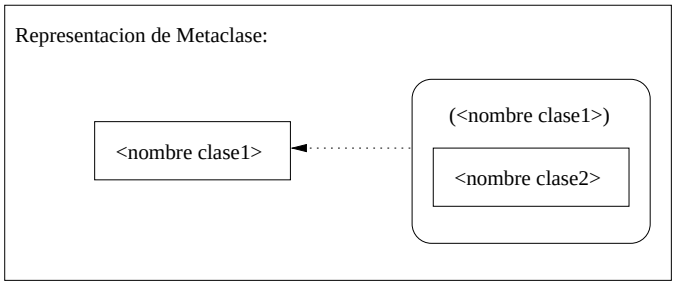

Figura 1.3: Representación de Metaclases.

1.2. MacOS

MacOS es el sistema operativo por defecto para las computadoras personales Macintosh. Su principal ventaja es apreciable desde el primer momento: La facilidad de uso.

Como sistema operativo provee los siguientes servicios (como se describe en [AC93] y [AC92]):

- **Manejador de Eventos**
  El Manejador de Eventos provee rutinas que informan acerca de las acciones ejecutadas por el usuario y notifica los cambios en el estado de procesamiento de la aplicación, adicionalmente provee rutinas que la aplicación puede usar para comunicarse con otras aplicaciones.

- **Manejador Gestalt**
  El Manejador Gestalt provee de una manera simple y eficiente para que la aplicación determine la configuración de hardware y software que posee el usuario en tiempo de ejecución, con el fin de explotar al máximo los recursos disponibles o informar al usuario la falta de algún recurso.

- **Utilidades de Manejo de Memoria**
  Las Utilidades de Manejo de Memoria proveen rutinas especializadas que pueden ser usadas para examinar o controlar ciertos aspectos del ambiente de memoria.

- **Manejador de Memoria**

El Manejador de Memoria es la parte del sistema operativo que controla la asignación dinámica del espacio en memoria.

- **Servicios de Multiprocesamiento**
  Los Servicios de Multiprocesamiento proveen rutinas que permiten a la aplicación crear hilos de ejecución separados llamados tareas. Las tareas son programadas preventivamente en el procesador disponible en el sistema, aún si solo se cuenta con un procesador.

- **Manejador de Notificación**
  El Manejador de Notificación provee rutinas que permiten a los programas que se ejecutan en segundo plano (*background*) comunicar información al usuario.

- **Manejador de Procesos**
  El Manejador de Procesos controla el acceso a los recursos compartidos y administra la programación y ejecución de aplicaciones, además permite que varias aplicaciones compartan tiempo de procesador y recursos del sistema.

- **Utilidades de Cola**
  Las Utilidades de Cola proveen rutinas que permiten manipular estructuras de datos tipo cola mantenidas por el sistema operativo; también puede ser usado para crear y manipular colas propias.

- **Manejador de Errores**
  El Manejador de Errores asume el control del sistema cuando ocurre un error y se encarga de mostrar cualquier mensaje de error al usuario.

- **Manejador de Hilos de Ejecución**
  El Manejador de Hilos de Ejecución provee una interfaz de programación para crear hilos de ejecución cooperativos en la aplicación.

- **Manejador de Tiempo**
  El Manejador de Tiempo permite que las aplicaciones y otros programas puedan programar rutinas de ejecución en el futuro. Provee un método independiente del hardware para llevar a cabo tareas relativas al tiempo.

- **Manejador de Memoria Virtual**
  El Manejador de Memoria Virtual es la parte del Sistema Operativo que per-

mite extender la memoria más allá del límite del espacio de direcciones que provee la memoria RAM disponible.

1.3. Linux

Linux es un sistema operativo tipo UNIX creado por Linus Torvalds con la ayuda de desarrolladores en todo el mundo. Está desarrollado bajo la GPL (*GNU General Public Licence*), por lo que el código fuente es gratis y está disponible para cualquier persona.

Este sistema operativo corre bajo Intel 386/486/Pentium, Motorola 680x0, Alpha DEC, Sun SPARC y arquitecturas PowerPC (incluyendo PowerMac, Motorola e IBM).

Entre las características más sobresalientes de Linux tenemos:

- Multitarea: Varios programas corriendo al tiempo.
- Multiusuario: Varios usuarios en el mismo computador al tiempo.
- Multiplataforma: Corre en diferentes arquitecturas.
- Protección de memoria entre procesos: De esta forma un programa no puede afectar a otros y provocar un fallo general del sistema.
- Enlace con librerías dinámicas y estáticas.
- Múltiples consolas virtuales: Varias sesiones independientes en la consola.
- Soporte para los sistemas de archivos más comunes, p.ej. FAT (MS-DOS/Windows) y HFS (Macintosh).
- Soporte para redes TCP/IP, incluye servicios como ftp, telnet, NFS, etc.
- Cuenta con una implementación de código abierto (distribuida bajo la GPL) redistribuible del sistema X-window denominada XFree86

2 OpenMusic

OpenMusic (OM) es un lenguaje de programación visual orientado a objetos basado en Common Lisp/CLOS. Es una aplicación de propósito general que provee un ambiente de ayuda a la composición musical implementando un conjunto de objetos representados por iconos que pueden ser arrastrados de un lugar a otro.

OpenMusic está dividido en dos partes: el *Kernel* y los *Projects*. En el Kernel están definidos los objetos principales tales como *Workspaces*, *Patches*, *Maquettes*, etc. (que se explicarán más adelante) y su manipulación gráfica. En los Projects se encuentran las definiciones de las clases especializadas (básicas y musicales) escritas directamente en Common Lisp. Sin embargo, OpenMusic permite al usuario crear sus propios proyectos y librerías siguiendo las especificaciones dadas en los capítulos 3 y 4 del manual de desarrollo de OM [Gro98]. En la figura 2.1 puede verse la estructura de OpenMusic (figura tomada de [Gro98]).

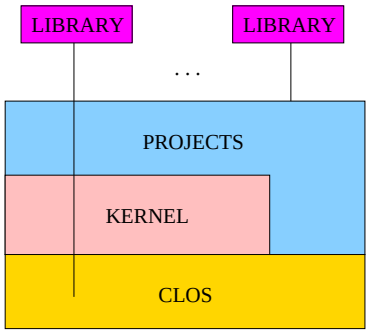

Figura 2.1: Estructura de OpenMusic.

2.1. Organización de Clases de OpenMusic

La clase raíz de OpenMusic es `OMObject` (todo objeto en OpenMusic es un `OMObject`).

Bajo `OMObject` se encuentra `OMBasicObject`, clase de los Meta-Objetos de OpenMusic (definiciones que son independientes de su visualización) tales como `OMPatch`, `OMClass` y `OMGenericFunction`.

`OMPersistantObject` es la clase del conjunto de Meta-Objetos que tienen un archivo asociado, es decir, objetos persistentes o que pueden ser salvados. Esta clase hereda de `OMBasicObject`.

La jerarquía de clases principales de OpenMusic se presenta en la figura 2.2 (adaptada de [Gro98]).

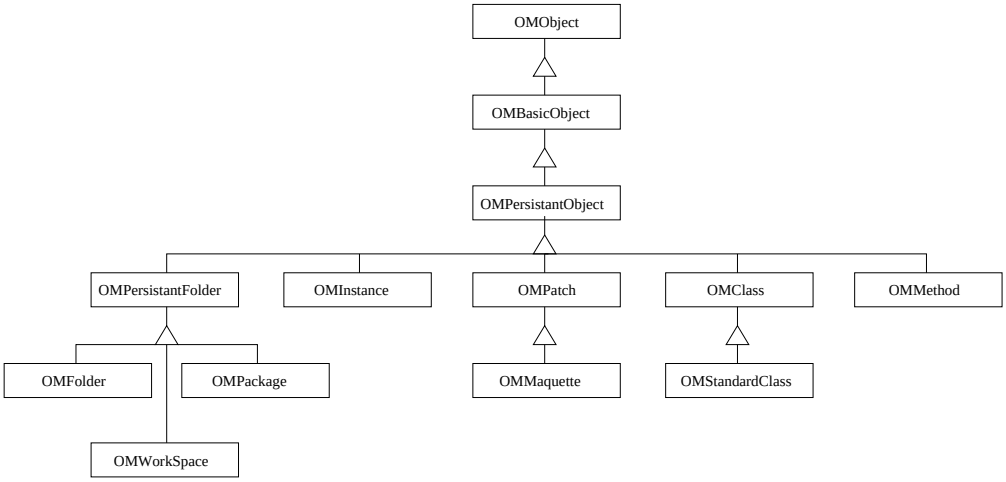

Figura 2.2: Herencia de Clases Principales de OpenMusic

Todos los Meta-Objetos de OM pueden ser visualizados como Iconos (*Simple Frames*) o como Contenedores (*Container Frames*). La clase `OMFrame` es la clase gráfica raíz.

La clase de los iconos, `OMSimpleFrame`, está dividida en: `icon-finder` y `boxframe`, con el fin de diferenciar los iconos tipo explorador (dibujo + nombre) de los iconos tipo caja (entradas + dibujo + nombre + salidas).

Un Contenedor, cuya clase base es `OMContainerFrame`, es un Meta-Objeto compuesto de una ventana, un editor y un panel dentro del cual se encuentra una lista de iconos.

La clase `OMBox` es la clase más general de los objetos que pueden ser conectados gráficamente. Existen dos subclases de `OMBox`: `OMBoxClass` y `OMBoxCall`. La primera es la clase de herencia, es decir, la conexión entre dos cajas representa una herencia entre ellos. `OMBoxCall` es la clase para cajas en un Patch, la conexión entre cajas representa una composición funcional.

La jerarquía de clases gráficas de OpenMusic se presenta en la figura 2.3 (adaptada de [Gro98]).

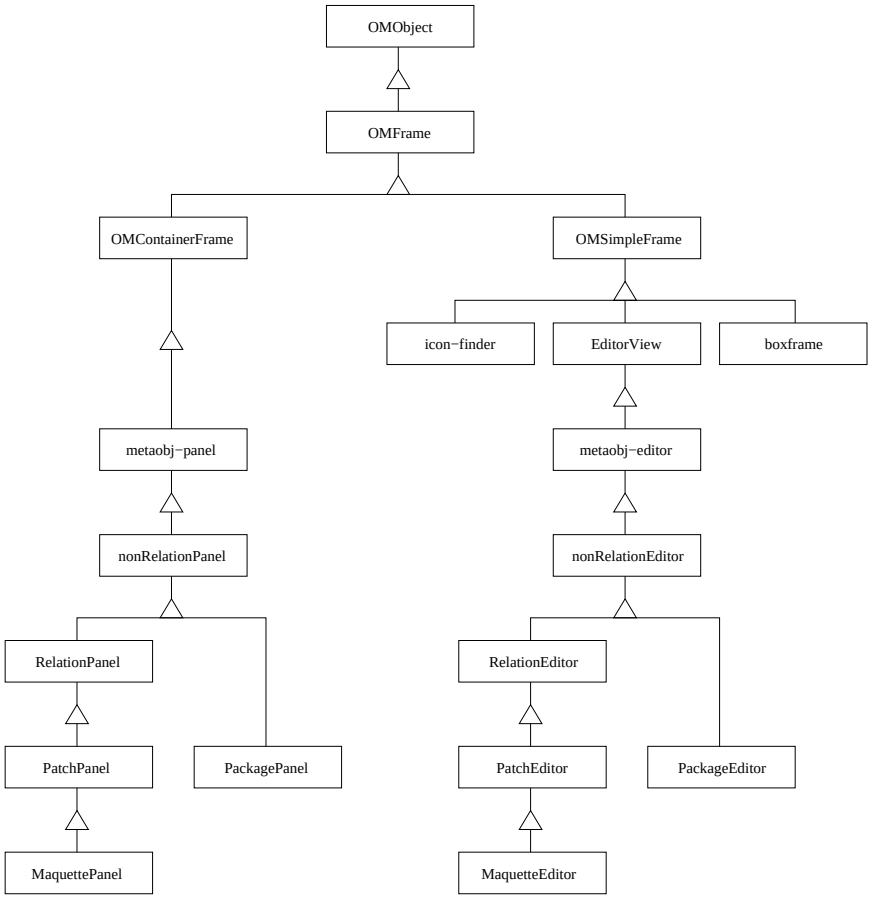

Figura 2.3: Herencia de Clases Gráficas de OpenMusic

2.2. Metaclases

En Common Lisp, las metaclases se usan para proveer formas particulares de optimización de objetos [KdRB91].

En OpenMusic, las metaclases se usan para permitir la programación visual, ya que todos los Meta-Objetos tienen como metaclase a `OMStandardClass` (no `standard-class`, la clase por defecto de CLOS), y ésta a su vez tiene como metaclase y superclase a `OMClass`. Es decir, las clases de OM por medio de su metaclase, pueden acceder atributos gráficos como iconos y contenedores.

2.3. Workspace

El Workspace es la ventana principal de OpenMusic donde se crean nuevos Patches, Maquettes y folders que contendrán Patches, Maquettes y subfolders. Cuando se presenta el evento de "doble-click" sobre el icono de un Patch o una Maquette, se crea una ventana que muestra su contenido.

En la figura 2.4 se muestra el Workspace con cuatro Patches (*patch*, *situation-nqueens*, *queen-cnstr* y *tutorial sit 5a*), una Maquette (el icono seleccionado *maquette*) y tres folder (*mikhail examples*, *packages* y *globals*). Es de notar que, tanto en el Workspace como en los folders, los iconos que se muestran son de tipo explorador (dibujo + nombre).

Dos folders especiales aparecen en la parte de abajo del Workspace: globals y packages. Globals es el folder donde se ponen las variables que se quieren compartir entre Patches. Packages es el lugar donde son almacenadas todas las funciones y clases que pueden ser arrastradas desde aquí hasta las ventanas de los Patches.

El Workspace tiene asociado la clase `OMWorkspace`. Dicha clase hereda de `OMPersistantObject` y tiene además los atributos de elementos que contiene (Patches, Maquettes o folders). También tiene los métodos de inicialización que cargan las variables globales, el folder de paquetes, las preferencias y todos los elementos del Workspace.

Al arrancar OpenMusic, se verifica que exista el folder correspondiente al Workspace del usuario. Si éste no exite, se crea uno nuevo con los elementos básicos para trabajar.

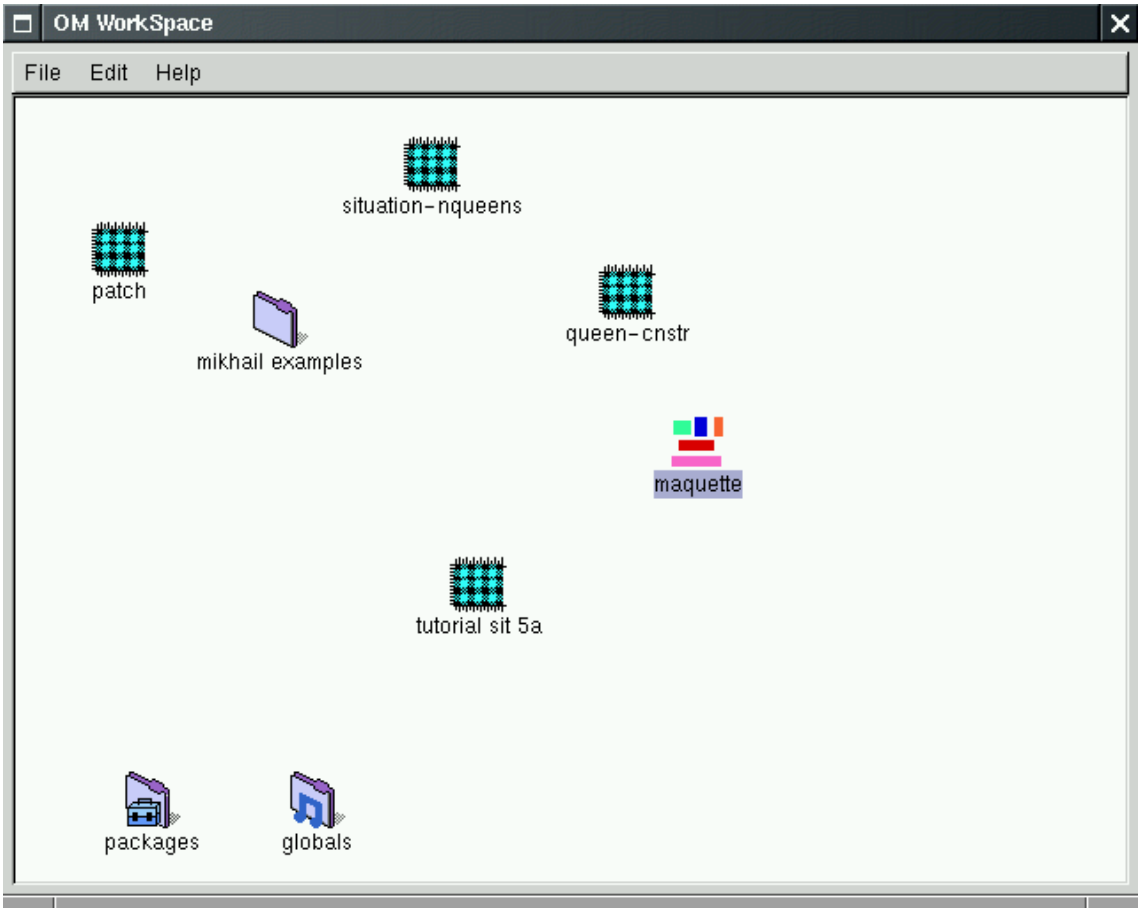

Figura 2.4: Workspace

2.4. Patches

Un Patch es la unidad básica de programación en OpenMusic. Se crea en el Workspace.

Cuando se presenta el evento de "doble-click" sobre un Patch, se abre un editor donde se construyen los programas visuales arrastrando objetos y conectándolos entre ellos. Dichos objetos pueden ser clases o funciones de algún paquete (arrastrados desde la ventana de paquetes, desde la barra de menús o desde otro Patch), Patches o Maquettes.

En la figura 2.5 se muestra el algoritmo para resolver el problema de las n-reinas. Los iconos en los Patches son de tipo caja (entradas + dibujo + nombre + salidas)

y pueden ser conectados haciendo "click" desde las salidas de una caja hasta las entradas de otra caja (nunca de las entradas a las salidas). Cada entrada de una caja representa un argumento de la función lisp asociada y las salidas son los valores que retorna la función.

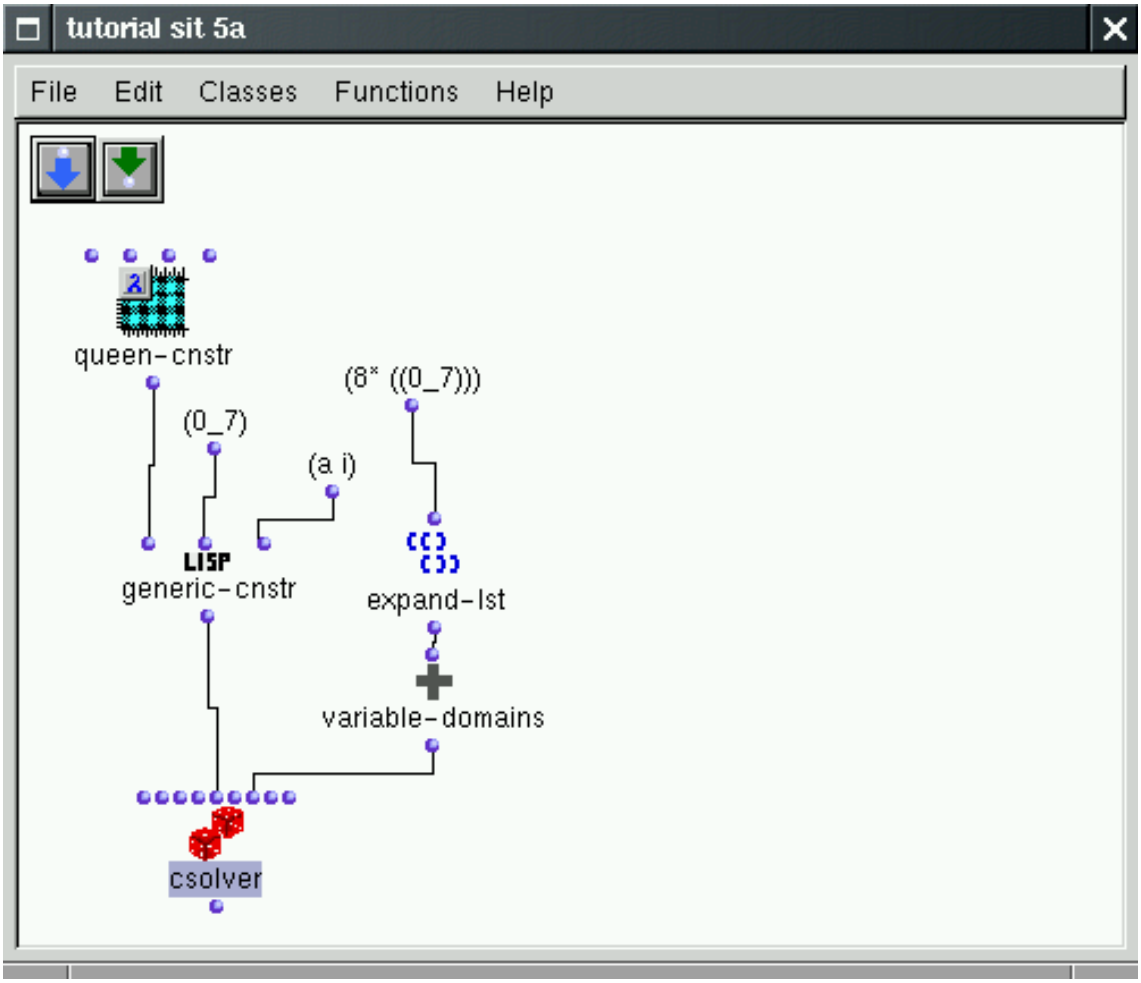

Figura 2.5: Patch

En el caso en que un Patch quiera ser usado como dato en otro programa (como el caso de la figura 2.5, donde *queen-cnstr* es un Patch dato dentro del Patch *tutorial sit 5a*), se pueden usar los botones de entradas y salidas, ubicados en la esquina superior izquierda del área de trabajo del editor de Patches.

La clase base de los Patches es `OMPatch` que hereda, al igual que `OMWorkspace`, de `OMPersistantObject`. Tiene como atributos propios: una lista de cajas, el código

lisp asociado a él (cada Patch tiene un código lisp con el cual se evalúan los datos), las conexiones entre las cajas, una bandera que dice si el Patch ya fué compilado, etc.

Cuando se salva un Patch, se llama al método `omNG-save`, y éste guarda en un archivo el código lisp correspondiente a la visualización gráfica, es decir, se guarda un archivo texto y no imágenes. En otras palabras, el código reconstruye el Patch gráficamente cuando se presenta el evento de "doble-click" sobre el icono, creando las instancias de las clases y llamando a los métodos de visualización.

2.5. Maquettes

Una Maquette es un tipo especial de Patch donde se tiene en cuenta el tiempo. Lo anterior quiere decir que los objetos que son arrastrados a una Maquette se organizan aquí de una forma cronológica. Dicho objetos "arrastrables" a una Maquette incluyen: un archivo midi, un Patch, otra Maquette, una variable desde la ventana `Globals` o una clase desde los paquetes.

La clase `OMMaquette` es una subclase de `OMPatch` que además trae una lista de parámetros para la inicialización y visualización de las maquettes.

El desarrollo de las maquettes no está dentro del alcance del trabajo de grado, por lo que en OpenMusic para Linux sólo se crean las instancias de la clase `OMMaquette` (se crea el icono en el Workspace) pero no puede abrirse el editor de ellas.

2.6. Packages

Los paquetes son colecciones de clases y funciones genéricas con las que se programa en OpenMusic. Ellas se encuentran en el folder `package` que se encuentra en la parte inferior del Workspace y en la barra de menús de los Patches.

Las librerías son un conjunto especializado de clases y funciones genéricas escritas directamente en Common Lisp.

La ventana de paquetes se puede apreciar en la figura 2.6. En ella se pueden ver los paquetes y clases propias de OpenMusic, además de las librerías de usuario adi-

cionales y los paquetes que el usuario haya creado. La visualización de la ventana está en forma de árbol ya que un paquete puede contener también paquetes. Para ver las funciones de un paquete se debe hacer "click" con el botón window (option en Macintosh).

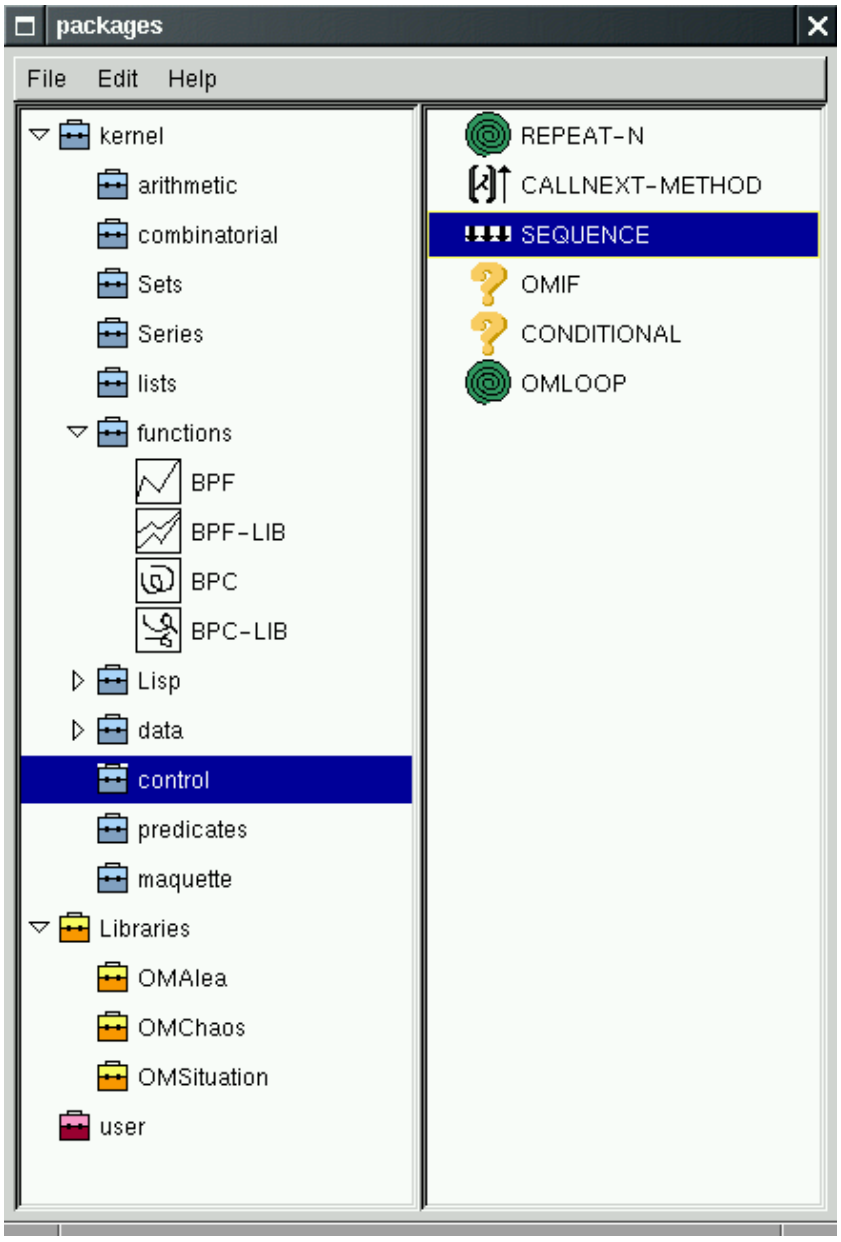

Figura 2.6: Packages

Clases y funciones pueden ser arrastradas de aquí (la ventana de paquetes) a los

Patches para ser usados.

Los paquetes pertenecen a la clase `OMPackage` quien hereda de `OMPersistantFolder` y tiene como principales atributos: los subpaquetes, las clases y las funciones que contiene. El paquete raíz que contiene todos los paquetes de OpenMusic esta instanciado en la variable `*om-package-tree*`. De ella se desprenden otras variables (`*package-user*`, `*control-package*`, `*maquette-package*`, `*lisp-package*`, etc.) que contendrán todas las funciones y clases.

Las librerías están asociadas a la clase `OMLib`, subclase de `OMPackage`. La razón de tener librerías y paquetes en clases diferentes es que las librerías son cargadas dinámicamente mientras que los paquetes no. Esto significa que para poder usar las clases y funciones de una librería, se debe hacer "doble-click" en la fila correspondiente, en la ventana de paquetes; caso que no corresponde en los paquetes ya que estas fueron cargadas al iniciar OM.

Cuando se inicia OpenMusic, antes de inicializar el Workspace, se cargan los paquetes y librerías mediante las funciones: `init-om-package` y `init-om-libs`. Estas funciones incluyen los paquetes y librerías como elementos en las variables de los paquetes.

3 Macintosh Common Lisp

En esta sección se describe el compilador[1] que se usó para crear OpenMusic en su versión original para Macintosh. Se describen las características más sobresalientes de Macintosh Common Lisp y que fueron relevantes en la implantación de la aplicación en Linux.

Macintosh Common Lisp (MCL) es un ambiente de programación flexible y fluido para el desarrollo de herramientas de software y aplicaciones, creado por Digitool Inc. Corre sobre Macintosh PowerPC con un sistema operativo MacOS 7.5 o superior.

Debido a que el Sistema Operativo MacOS es totalmente gráfico, MCL trae consigo todos las funciones, componentes y eventos que son posibles de manejar en dicho sistema como clases propias.

Las características más importantes del compilador MCL se mostrarán a continuación. Cada característica que se enumera se debió reimplementar en la versión de OM para Linux, ya que éstas no las posee el compilador de Lisp escogido para el porte y OpenMusic para Macintosh las usa.

3.1. Puntos

Un punto en MCL es un simple entero usado como codificación de una coordenada con componentes x y y, con el propósito de ahorrar espacio.

Para definir un punto se tiene la macro `#@`, la cual toma dos argumentos correspondientes a la coordenada x y la coordenada y, y los codifica generando un entero que

[1] Las implementaciones de Common Lisp permiten interpretar y/o compilar en bytecodes que pueden ser interpretados posteriormente. En el presente trabajo se hará referencia a cualquier implementación de Common Lisp usando el término "Compilador".

representa el punto con coordenada horizontal x y coordenada vertical y.

Por ejemplo, `#@(30 -50)` expande a -3276770, el entero que representa el punto con coordenadas x=30 y y=-50.

3.2. Vistas y Ventanas

MacOS emplea un sistema de vistas para dibujar y mostrar los gráficos.

La clase más general es `simple-view`, la cual es la clase de vistas que no contienen subvistas; por esta razón las vistas-simples pueden ser fácilmente dibujables. De ésta clase heredan todas las vistas. En ella se encuentran los atributos de posición, tamaño, tipo de fuente, vista donde se encuentra (contenedor), puntero a la ventana que lo contiene, ayuda, etc. Ejemplos de vistas-simples son los *radio buttons* y los *checkboxes*.

La subclase más importante de `simple-view` es `view`. Ella hereda todos los atributos de las vistas-simples y además tiene una lista de las subvistas que tiene. La mayoría de operaciones gráficas en MacOS se definen sobre vistas teniendo en cuenta su propio sistema de coordenas (el cual tiene como punto origen (0,0) en la esquina superior izquierda), sus fuentes y la actividad con el *mouse*.

Las ventanas son subclases de `view`, que pueden contener vistas, pero que ellas mismas no pueden ser subvistas, es decir, una ventana no puede estar contenida en otra ventana. La clase base de las ventanas es `window`, que tiene todas las características de cualquier ventana. `window` tiene todos los atributos de `view` más el título de la ventana, una bandera que dice si la ventana se está mostrando o está oculta, el tipo de ventana, etc.

Todo lo anterior se puede entender como que en el Sistema Operativo MacOS se tiene una vista principal (toda la pantalla). En la pantalla pueden haber ventanas y dentro de éstas hay vistas que contienen vistas-simples . Además, las ventanas no pueden tener ventanas dentro de sí mismas.

La jerarquía de clases gráficas de MCL se presenta en la figura 3.1 (Adaptado de [Dig96]).

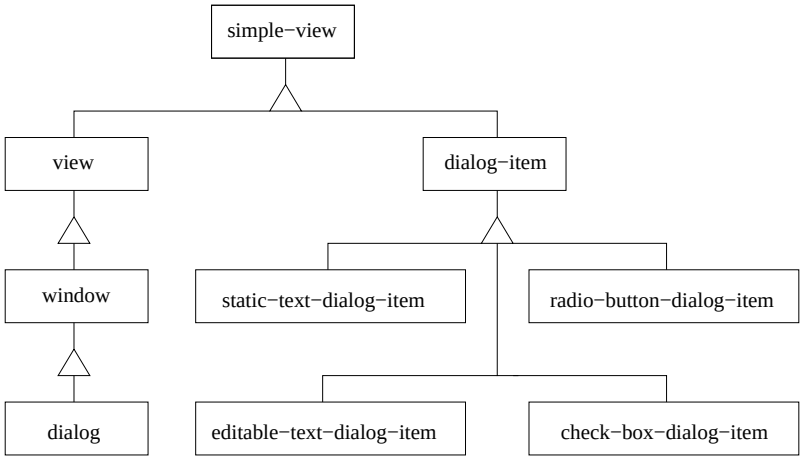

Figura 3.1: Herencia de Clases Gráficas de MCL

3.3. Diálogos

En MacOS, la comunicación entre el usuario y el software se hace mediante *dialog items* que lanzan procedimientos y se encuentran dentro de una vista. La clase base de los dialog items es la clase abstracta `dialog-item` que hereda directamente de `simple-view`. Esta clase no se instancia directamente, solamente alguna de sus subclases:

- `button-dialog-item`
- `editable-text-dialog-item`
- `static-text-dialog-item`
- `check-box-dialog-item`
- `radio-button-dialog-item`
- `table-dialog-item`
- `scroll-bar-dialog-item`
- `sequence-dialog-item`

Cada clase de dialog items tiene los atributos de `simple-view`, el texto del dialog item, el estado del item, la acción que realiza y los respectivos atributos especializados de cada subclase (i.e. botón por defecto para los `button-dialog-item`).

Los cuadros de diálogo son subclases de `window` que contienen dialog items en una forma estructurada para controlar acciones bien definidas. Existen dos tipos de cuadros de diálogo: los modales y los no modales. Los cuadros de diálogo modales son aquellos que deben ser cerrados para que se puedan realizar otras acciones en el programa; los no modales son por el contrario, cuadros de diálogo que aparecen en la pantalla pero no interfieren con otras acciones de la aplicación.

3.4. Menús

Un menú en MCL es una lista de menú items (los cuales pueden ser menús también) que pueden ser instalados o desintalados en la barra de menús. Menús, menú items y la barra de menús son instancias de las clases `menu`, `menu-item` y `menubar` respectivamente, y éstas heredan de la clase abstracta `menu-element`.

La barra de menús es única para todas las ventanas en MacOS, esto quiere decir que solo puede mostarse una barra de menús a la vez. De allí que si se quiere compartir una barra de menús para varias ventanas, solo sea necesario declarar una variable global que compartirán éstas.

Los menús pueden ser creados en cualquier momento sin necesidad de instalarlos en la barra de menús o en otros menús (caso en el que son tratados como menú items). La clase `menu` tiene los atributos de título, items que contiene y ayuda. MacOS tiene el menú especial *Apple* que no puede ser borrado de la barra de menús y al cual no se le pueden remover algunos menú items.

Los menú items son el cuerpo de los menús. Ellos tienen asociados acciones. Los atributos de la clase `menu-item` incluyen el menú donde está instalado, el título, el acceso directo por teclado, la acción asociada, la bandera que dice si ese item está desabilitado, el estilo, la ayuda, etc. Cuando se quiere crear un separador en un menú, simplemente se crea un menú item con titulo "-".

3.5. Eventos

Los eventos son interrupciones a los programas que realizan una acción y además pueden requerir una respuesta. MCL maneja los eventos en procesos diferentes y la ejecución del programa no sigue hasta que dichos procesos retornen un dato.

Cuando un evento ocurre, MCL toma del Sistema Operativo dicho evento, determina de que tipo es y llama la función correspondiente que debe haber sido definida en el código fuente. Éstas funciones deben finalizar con la cadena "-event-handler" y comenzar con el tipo de evento respectivo:

- `view-click-event-handler`
- `view-key-event-handler`
- `view-activate-event-handler`
- `view-deactivate-event-handler`
- `view-mouse-enter-event-handler`
- `view-mouse-leave-event-handler`
- `window-select-event-handler`
- `window-key-up-event-handler`
- `window-mouse-up-event-handler`
- `window-close-event-handler`

MCL tiene también funciones que ofrecen información importante a los eventos:

- `view-mouse-position`: Posición actual del *mouse*.
- `mouse-down-p`: Si se hizo click.
- `double-click-p`: Si se hizo doble click.
- `command-key-p`: Si se pulsó la tecla *command*.

- `control-key-p`: Si se pulsó la tecla *control*.
- `option-key-p`: Si se pulsó la tecla *option*.
- `shift-key-p`: Si se pulsó la tecla *shift*.

3.6. Recursos

Un recurso es un dato de cualquier tipo guardado en la zona de recursos de un archivo. Éstos son elementos básicos que describen menús, ventanas, controles, cuadros de diálogo, sonidos, fuentes e iconos. Cada aplicación interpreta los datos de los recursos de acuerdo a su tipo de recurso.

En MacOS, un archivo está dividido en el área de datos y el área de recursos. El área de datos contiene los datos creados por el usuario. El área de los recursos de un archivo contiene la cabecera de los recursos, los recursos mismos y el mapa de los recursos (ver figura 3.2 - tomada de [AC93]).

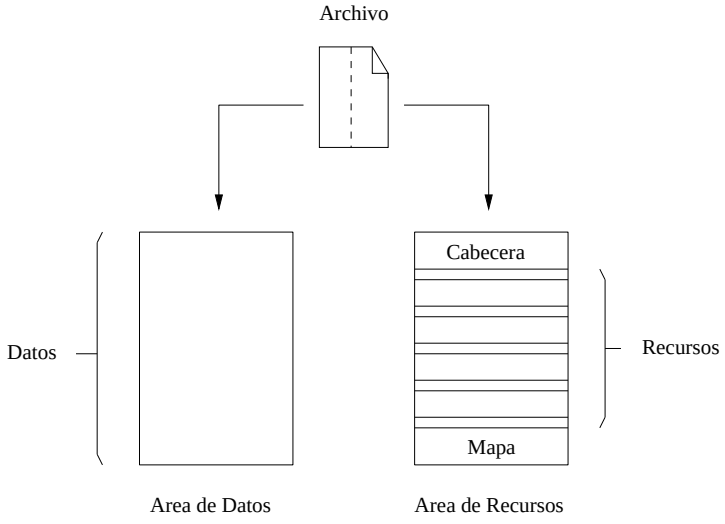

Figura 3.2: División de los archivos en MacOS

La cabecera tiene la dirección del comienzo de los recursos y el mapa. El mapa contiene la información de los recursos y la dirección de cada uno de ellos.

Cuando se crea un recurso, se le asigna un tipo y un identificador. El tipo de un recurso es una secuencia de cuatro caracteres que identifican de forma única un tipo

específico de recurso, mientras el identificador es un número que identifica un recurso específico. Los tipos de recursos más usados son:

- `CODE`: Segmentos de código de la aplicación.
- `CURS`: Cursores del *mouse*
- `DLOG`: Cuadros de diálogo.
- `ICN#`: Iconos con máscara.
- `ICON`: Iconos sin máscara.
- `MBAR`: Barras de menús.
- `MENU`: Menús
- `NFNT`: Fuentes
- `WIND`: Ventanas

4 Herramientas de Desarrollo en Linux

El presente capítulo muestra las herramientas de desarrollo que se escogieron para desarrollar el porte de OpenMusic en Linux. Inicialmente se cubre la parte de escogencia del compilador, se presenta el compilador escogido, sus características y la razón por la cual se escogió. Posteriormente se aborda la escogencia de la librería para la interfaz gráfica, se muestran las librerías gráficas disponibles para la implantación en Linux, la librería escogida, sus características y el motivo por el cual se tomó como librería gráfica para el porte.

4.1. Compilador

Para el porte de OpenMusic a Linux se necesitaba un compilador de Common Lisp que cumpliera con tres requerimientos básicos:

1. Proveer una implementación de ANSI Common Lisp.
2. Proveer una implementación de CLOS.
3. Estar cubierto por una licencia de uso libre, preferiblemente la GPL.

Las cuatro principales implementaciones de ANSI Common Lisp para Linux son:

- CLISP
- CMUCL

- Allegro Common Lisp
- Harlequin LispWorks

Las dos últimas (Allegro Common Lisp y Harlequin LispWorks) son implementaciones comerciales, por tanto al no cumplir con el tercer requerimiento fueron descartadas.

Se optó entonces por evaluar las dos primeras opciones (CLISP y CMUCL)

En un pricipio se decidió usar CLISP, ya que existe un porte de este compilador a Linux/PowerPC (siendo PowerPC la arquitectura nativa de OpenMusic); sin embargo, luego de probarlo por algún tiempo sobre la platforma Linux/x86 se notó que al correr una aplicación con la librería gráfica escogida (CLG), el *listener* de Lisp[1] no quedaba libre, es decir, sólo se podía usar funciones y clases de OpenMusic y no se podían definir en interpretar funciones adicionales. Esto rompía totalmente con la intención de ser una aplicación de propósito general, es decir, una aplicación donde no sólo se puedan representar estructuras musicales o de algún paquete propio de OM, sino que se puedan crear funciones, clases, métodos, etc. en cualquier momento, se puedan usar y, si se quiere, se puedan visualizar gráficamente.

Adicionalmente el porte del compilador de Clisp a la plataforma Linux/PPC carece de soporte para uso de código no nativo en Lisp, motivo por el cual las librerías para interfaz gráfica disponibles en otros lenguajes(p.ej. C, C++) no podían ser usadas en este compilador. Se abandonó entonces el uso de ese compilador y se optó por usar un compilador que fuera capaz de proveer todo el soporte necesario para OpenMusic.

Las pruebas arrojadas por el compilador CMUCL sobre la plataforma Linux/x86 fueron completamente satisfactorias: El *listener* de Lisp quedaba libre una vez que la aplicación OpenMusic se encontraba corriendo, lo cual permitia hacer uso de las funciones y clases de OpenMusic y además de cualquier función o clase en Lisp definida por el usuario.

[1]El listener de Lisp es el interpretador que siempre está presente en la compilación del código y que permite definir nuevas funciones y clases en tiempo de ejecución.

4.1.1. CMUCL

CMU Common Lisp o CMUCL (http://cons.org/cmucl/) es una implementación del lenguaje de programación Common Lisp, desarrollada en el Departamento de Ciencias de la Computación de la Universidad Carnegie Mellon.

CMUCL incluye una interface con código no nativo (*foreign code*) y acceso a métodos predefinidos para llamados al sistema Unix.

Este compilador corre bajo Intel x86/FreeBSD, Intel x86/Linux (libc5 y glibc2), SPARC/SunOS, PA-RISC/HPUX, DEC Alpha/OSF1 (Digital Unix Compaq Tru64 Unix) y SGI/Irix6.

Por ser CMUCL un software de dominio público, los usuarios pueden modificar los fuentes y/o enlazarlos con otros objetos; además corre con prácticamente todos los paquetes de Common Lisp gratis como CL-HTTP, Garnet y CLIO/CLOE.

CMUCL trae una gran cantidad de paquetes. Los paquetes usados en el presente trabajo de grado fueron:

- pcl: Portable Common Lisp, una implementación portable de CLOS.
- extensions (ext): Extensiones locales de Common Lisp
- common-lisp (cl lisp): Contiene todos los simbolos definidos por *Common Lisp: The Language* [Ste90].
- unix: Interfaz al sistema UNIX.
- system (sys): Contiene funciones e información necesaria para el sistema.
- common-lisp-user (user cl-user): El paquete por defecto. Donde se guarda el código del usuario y los datos.

Adicionalmente incluye dos paquetes para generar interfaces gráficas: CLX y CLM.

4.2. Librería para Interfaz Gráfica

Una vez escogido el compilador para el porte era necesario encontrar una librería para generar la interfaz gráfica de la aplicación que cumpliera con los siguientes requerimientos:

1. Proveer un conjunto de componentes visuales similares a los de MCL.
2. Poder ser usada con el compilador de Common Lisp escogido.
3. Contar con un soporte para desarrollo adecuado.

El sistema X-window, desarrollado a mediados de los 80's, provee el manejo y exposición de una interfaz gráfica de usuario para los sistemas operativos tipo UNIX. A diferencia del manejador de ventanas de Microsoft Windows o MacOS (quienes muestran las aplicaciones gráficas de una forma local en el PC), el protocolo X distribuye los procesos de las aplicaciones de una forma cliente-servidor al nivel de la aplicación.

Linux utiliza una implementación de código abierto (*open source*) redistribuible del sistema X-window denominada XFree86 (http://www.xfree86.org/).

Las librerías gráficas más usadas para el desarrollo de aplicaciones en Linux son las siguientes:

- **Xlib y Xaw**
  Xlib (http://www.x.org/) es la librería más generica y de más bajo nivel de X-window, la cual contiene todas las funciones asociadas con X. La librería Xaw contiene el conjunto de componentes visuales (*widgets*) de X.
- **Motif (LessTif)**
  Motif es una herramienta basada en Xlib que ofrece, principalmente, un conjunto básico de componentes visuales. Es desarrollado por Open Group, se incluye en la mayoría de sistemas Unix de tipo comercial. El Manejador de ventanas más conocido desarrollado usando motif es *Common Desktop Environment* (CDE).
  El proyecto LessTif (http://www.lesstif.org/) inició en 1994 por Chris Toshok

y *The Hungry Programmers* para proveer a la comunidad del software libre de una implementación grátis de la librería estándar Motif. El propósito de LessTif es alcanzar una total compatibilidad con las versiones comerciales de Motif para que pueda ser usado como un reemplazo de Motif.

- **Qt**
  Qt(tm) (http://www.trolltech.com/) es una herramienta para creación de interfaces gráficas de usuario escrita en C++ y completamente orientada a objetos.
  Qt es una herramienta multiplataforma cuyas aplicaciones pueden correrse tanto en el sistema X-Window (Unix/X11) como en Microsoft Windows NT y Windows 95/98 (el código fuente debe ser recompilado para cada plataforma deseada). Qt es la base del manejador de ventanas KDE.

- **Tcl/Tk**
  Tcl (http://www.scriptics.com/) provee un ambiente de programación portable para Unix, Windows y Macintosh que soporta procesamiento de cadenas y reconocimiento de patrones, acceso al sistema de archivos nativo, soporte para redes TCP/IP, y las construcciones tradicionales de programación como variables, ciclos, procedimientos, etc.
  Tk provee una interfaz gráfica de usuario sobre Unix, Windows y Macintosh, con un poderoso conjunto de componentes.

- **Garnet**
  Garnet (*Generating an Amalgam of Real-time, Novel Editors and Toolkits* - http://www.cs.cmu.edu/afs/cs.cmu.edu/project/garnet/www/) es un conjunto de herramientas escritas en Common Lisp usadas para crear interfaces gráficas de usuario. Fue creada en la Escuela de Ciencias de la Computación de la Universidad Carnegie Mellon y puede ser usada tanto en los sistemas Unix corriendo X como en Mac. Corre en casi todos los ambientes de Common Lisp tales como Allegro, Lucid, Clisp, Cmu y MCL; y en diferentes arquitecturas como Sun, DEC, HP, IBM, SGI, PCs corriendo Linux y Mac.
  Garnet no usa el estándar CLOS pero tiene una gran cantidad de características: la herramienta *Lapidary* para crear aplicaciones gráficas y widgets sin tener que programar, las restricciones integradas con los objetos del sistema, los objetos son refrescados automáticamente cada vez que ellos cambian, generación

de PostScript para cualquier imagen en la pantalla, etc.

- **GTK+, GDK y Glib**
  GTK+ (*the Gimp ToolKit* - http://www.gtk.org/) es un pequeño y eficiente conjunto de componentes visuales y funciones para su manipulación que contiene los componentes más comunes para creación de interfaces gráficas y, adicionalmente, otros más complejos como selección de archivos (*file selection*) y selección de colores (*color selection*). También provee características únicas como, por ejemplo, botones que pueden contener *widgets*, lo que permite incorporar mapas de bits, imágenes, etiquetas o cualquier combinación que el programador desee. Esta flexibilidad es adherida a la librería. Gtk+ requiere GDK y Glib.
  GDK (*the Gimp Drawing Kit*) es una librería diseñada como una cubierta para el llamado de funciones de Xlib. Realiza la mayoría de operaciones comunes y deseadas por un programador para dibujar sobre ventanas, sin que éste tenga que interactuar directamente con Xlib.
  Glib (*G Library*) es una librería escrita en C diseñada para resolver los problemas de portabilidad reemplazando algunas funciones estándar de *libc* tales como *malloc*. Provee funciones y definiciones para el uso de las aplicaciones GDK y GTK. También provee rutinas para el manejo de listas simples y doblemente encadenadas, funciones de error, analizador léxico, manejo de cadenas, entre otras.

Al evaluar la posibilidad de uso de cada una de estas librerías para el porte se obtuvieron los siguientes resultados:

Los compiladores de Lisp para Unix incluyen siempre el paquete CLX, correspondiente a la librería XLib. Esta es la librería gráfica de mas bajo nivel para generar interfaces gráficas; no posee un conjunto de componentes muy especializado, por tal razón el usuario debe implementarlos a partir de funciones gráficas muy primitivas. Escoger esta librería para el porte de OpenMusic implicaba considerar también el desarrollo de cada uno de los componentes necesarios, labor que resultaría demasiado dispendiosa e innecesaria, si se tiene en cuenta que existen otras librerías sí que incluyen un conjunto básico de componentes especializados; se optó entonces por desechar esta opción.

CMUCL incluye la libreria CLM, que corresponde a la librería Motif. Esta librería no es de tan bajo nivel como CLX, puesto que incluye un conjunto de componentes básicos, pero para el propósito de desarrollar OpenMusic era requerido un conjunto más amplio de componentes que podía ser proporcionado por otras librerías.

La librería Qt no provee actualmente un paquete en Lisp, por tanto no podía ser usada en el proyecto a menos que fuera implementado el paquete de interfaz de esta librería para Common Lisp, esta labor resultaba también dispendiosa e innecesaria teniendo en cuenta que otras librerías ya proveen un paquete en Lisp.

Tcl/Tk fue desechada desde un comienzo por recomendación del director del proyecto, debido a su ingrata experiencia en un proyecto anterior con características muy similares; lo anterior a pesar de que existe un paquete de interfaz en Common Lisp para esta librería y que es una de las librerías gráficas más portables y estandarizadas.

A simple vista, Garnet parece tener las características necesarias para generar la interfaz gráfica para OpenMusic, puesto que incluye un conjunto significativo de componentes y manejo de eventos, pero examinando un poco más a fondo este paquete y sus características se encontraron varias desventajas significativas. Entre las desventajas principales se encontraban:

- El desarrollo de Garnet había sido suspendido hace mas de cinco años, factor que imposibilitaba tener el soporte adecuado a la hora de desarrollar una aplicación compleja.
- El manual de referencia de Garnet menciona diversas características del paquete que en la práctica se comportan de una manera distinta.

CLG es un paquete que provee la interfaz de GTK+ para Common Lisp. GTK+ es una librería gráfica de alto nivel y con un conjunto considerable de componentes bastante similares a los componentes provistos por la librería gráfica de MCL; además tanto CLG como GTK+ se encuentran en desarrollo activo y cuentan con las siguientes fuentes de referencia:

- Soporte directo del autor de CLG: El autor del paquete y desarrollador principal del mismo puede ser contactado directamente por correo electrónico.

- Manual de referencia de GTK+ [Tea01]: Es un documento actualizado y que provee de una descripción para todas y cada una de las funciones de la librería misma.
- Tutorial de GTK+ [GM01]: Es un documento que permite al usuario principiante de la librería empezar a hacer uso de la misma mediante códigos de ejemplo sencillos.
- Lista de discusión de GTK+: Permite tener acceso a soporte directo de los desarrolladores de la librería, así como de miles de usuarios de la misma.

En conclusión GTK+ ofrece todas las características buscadas en una librería de interfaz gráfica para el desarrollo de OpenMusic.

4.2.1. Common Lisp Gtk+

Common Lisp Gtk+ (CLG) es un paquete de Common Lisp creado para CMUCL y CLisp por Espen S. Johnsen (espejohn@online.no). El objeto de este paquete es el manejo de las librerías de Gtk+ (the Gimp ToolKit) desde Lisp; este manejo incluye la posibilidad para el usuario de definir en Lisp variables y objetos de GTK así como la manipulación de los mismos mediante funciones también en Lisp.

GTK+ provee un sistema completo orientado a objetos implementado en C, con herencia, chequeo de tipos y una infraestructura para señales y punteros a funciones (callback functions).

CMUCL incluye el paquete pcl, que es una implementación portable de CLOS. CLOS sirve como punto de partida para la implementación de los vínculos de GTK+ para Common Lisp puesto que así la jerarquía de clases de GTK+ puede ser declarada tal cual en Lisp (siguiendo un esquema orientado a objetos), los tipos básicos de GTK+ y GLib tambien son declarados como tipos externos en Lisp.

La declaración de las clases de GTK+ en Lisp incluye la declaración de algunos atributos para cada clase. El acceso a estos en C suele hacerse mediante el uso de funciones especifícas o referenciando directamente la estructura correspondiente a la clase misma (las clases de GTK+ en C son implementadas como estructuras). El acceso en Lisp a estos mismos atributos es mucho mas simplificado y orientado a

objetos, ya que estas estructuras "simulan" ser clases en C mientras que en Lisp son declaradas y manipuladas como clases.

Las funciones principales para creación y manipulación de cada uno de los objetos en GTK+ deben ser declaradas como funciones foráneas en Common Lisp[2], esto es posible habiendo declarado de antemano las clases y tipos básicos de GTK+. Algunas estructuras que no corresponden directamente a una clase en GTK+ pueden ser manipuladas directamente en C mediante la sintaxis de manipulación de estructuras, para llevar a cabo la misma tarea en Lisp es necesaria la creación de ciertas funciones especiales en C que accesen estas estructuras y que retornen o modifiquen el valor correspondiente al campo especificado, estas funciones a su vez deben ser declaradas como funciones foráneas en Lisp.

Al generar una interfaz gráfica usando GTK+ para un programa en Common Lisp se sigue un esquema bastante similar al que se usa para la misma tarea para un programa en C. El esquema en Common Lisp es como sigue a continuación:

Consideraciones iniciales.

- Se definen en primera instancia las interfaces de interacción inicial con el usuario.
- A continuación se definen las interfaces de entrada y salida de datos.
- Para cada interfaz se debe definir los eventos que se desean manejar dentro de la misma y el tratamiento asignado a cada evento.
- Para cada interfaz y sus respectivos eventos se debe considerar cuáles son los componentes más adecuados para visualización, entrada y salida de datos y captura de eventos, GTK+ incluye un conjunto de componentes que suelen suplir la mayoría de las necesidades para una amplia gama de aplicaciones de propósito general.

Una vez se han seguido las consideraciones iniciales se procede a desarrollar el código para la interfaz gráfica.

[2]Funciones foráneas son aquellas funciones implementadas en otros lenguajes de programación que, al ser declaradas en Common Lisp, pueden ser usadas.

El desarrollo del código GTK+ para la aplicación debe ser llevado a cabo de una manera jerárquica, primero se define una ventana principal que permita incorporar un contenedor inicial. Los contenedores en GTK+ agrupan varios componentes; la manera en que estos quedan distribuidos dentro del contenedor depende del tipo de contenedor usado y, en algunos casos, de las propiedades modificables del mismo (i.e. ocupar todo el espacio horizontal y/o vertical disponible en el contenedor, centrar los componentes, etc.). Un contenedor puede agrupar también otros contenedores y estos a su vez pueden agrupar a otros; en última instancia, lo que se persigue es dar a la interfaz la apariencia deseada por el desarrollador.

Los componentes visuales utilizados durante la implementación del porte se detallan a continuación.

- **Ventana (GtkWindow)**
  Es el contenedor principal para los demás componentes, la Ventana es la encargada de reservar (*allocate*) el espacio dentro del servidor de ventanas X. En la figura 4.1 se puede apreciar el uso de este componente como la ventana de un Workspace en OpenMusic; este componente también fué usado como ventana de Folders, Patches, Packages y otros elementos de OpenMusic que requieren su propia ventana.

- **Caja Horizontal y Vertical (GtkHBox - GtkVBox)**
  Son contenedores que deben ir dentro de una ventana, permiten organizar componentes en su interior en una secuencia horizontal o vertical (dependiendo del tipo de caja usado), la Caja Vertical puede ser apreciada en la figura 4.1 con la ventana como su contenedor; en OpenMusic las Cajas fueron usadas como contenedores para cualquier conjunto de componentes hijos que requirieran una disposición secuencial horizontal o vertical (p.ej. Menús y Ventanas de Desplazamiento en el caso de la ventana del Workspace, ver figura 4.1).

- **Barra de Menús (GtkMenuBar)**
  Es un componente que permite agrupar uno o varios componentes del tipo Item de Menú (GtkMenuItem), lo cual produce como resultado un barra de menús tradicional. En la figura 4.2 se muestra una Barra de Menús con algunos Items. Las Barras de Menús fueron usadas en OpenMusic como las barras de menús para las ventanas de Workspaces, Folders, Patches, etc.

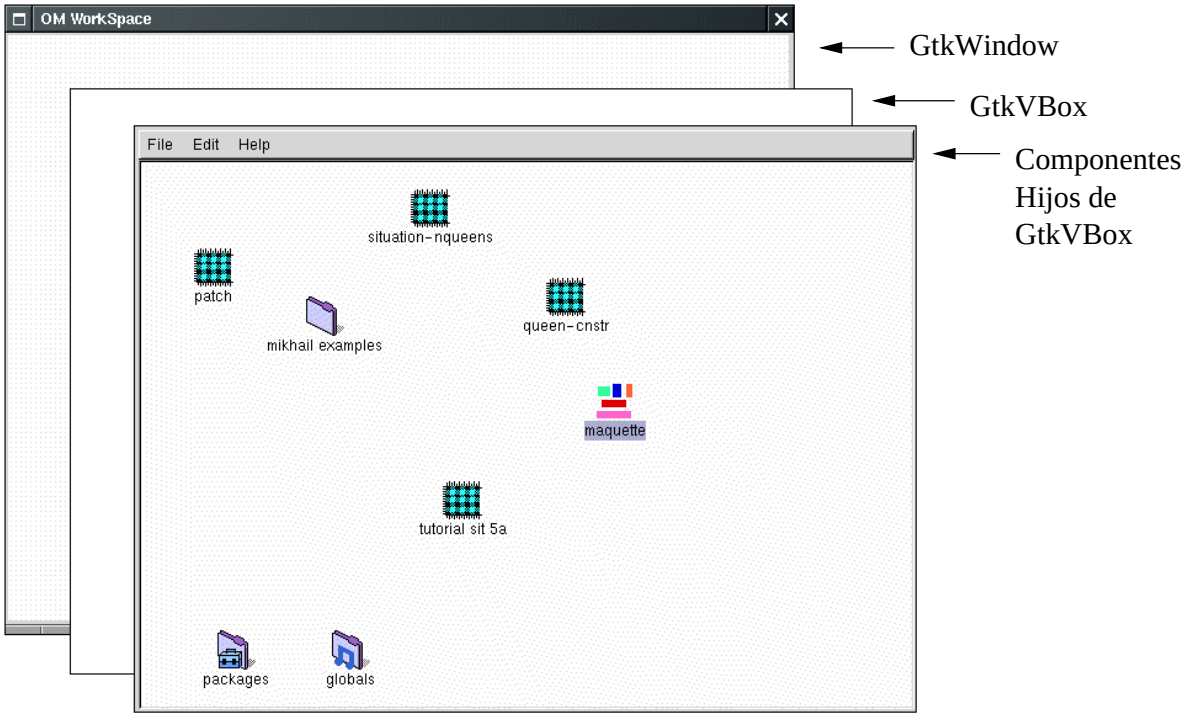

Figura 4.1: GtkWindow - GtkVBox

- **Menú (GtkMenu)**
  Es un componente que implementa un menú en cascada compuesto por una lista de componentes de tipo Item de Menú (GtkMenuItem) que pueden ser navegados y activados por el usuario. El desplazamiento en cascada de este tipo de componente es activado por un Item de Menú que haga parte de una Barra de Menús. La figura 4.3 muestra un Menú una vez que ha sido desplegado por el usuario al hacer click sobre el Item que lo activa.

- **Item de Menú (GtkMenuItem)**
  Es el componente usado para poblar los Menús y las Barras de Menús, se encarga de manejar correctamente la selección, alineación, eventos y submenús. Las figuras 4.2 y 4.3 muestran el uso de Items en Barras de Menús y en Menús respectivamente.

- **Ventana de Desplazamiento (GtkScrolledWindow)**
  Es un contenedor que solo puede contener un componente hijo, su función es proporcionar barras de desplazamiento a su componente hijo en la medida

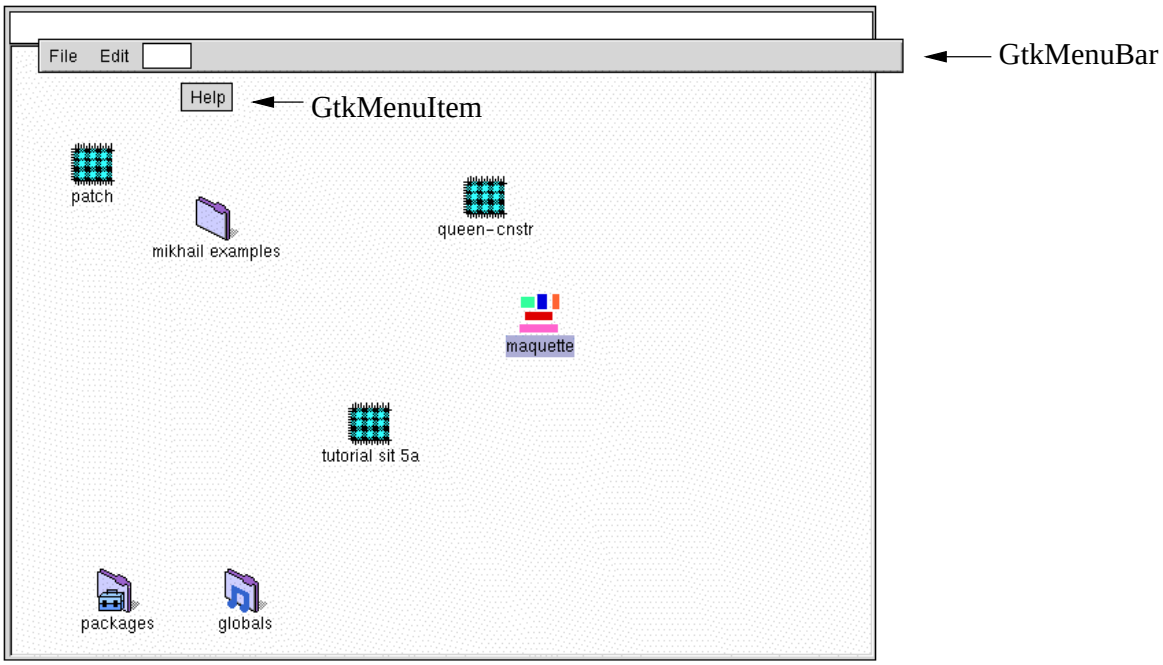

Figura 4.2: GtkMenuBar - GtkMenuItem

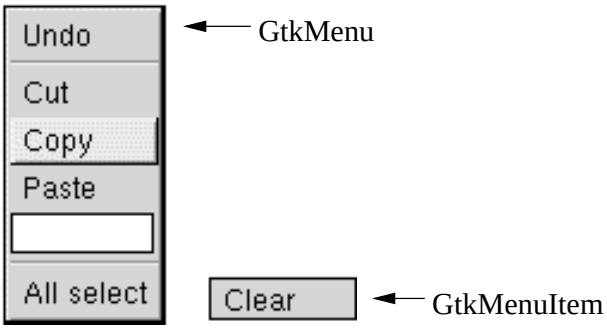

Figura 4.3: GtkMenu - GtkMenuItem

que sean requeridas. La figura 4.4 muestra un Ventana de Desplazamiento que con un Contenedor XY en su interior; este tipo de contenedor fué usado en OpenMusic para cualquier tipo de componente cuyo tamaño fuera mayor que el de la ventana que lo contiene, para permitir asi extenderlo y recorrerlo sin necesidad de ampliar la ventana.

- **Contenedor XY (GtkLayout)**
  Es un contenedor que permite organizar varios componentes dentro de sí mismo

especificando la posición x,y para cada uno de ellos. El tamaño máximo de un Contenedor XY es lo suficientemente grande como para considerarlo infinito, también posee la propiedad de redibujarse a sí mismo, incluidos sus componentes hijos. En la figura 4.4 se muestra un Contenedor XY dentro de una Ventana de Desplazamiento, el cual fue usado en OpenMusic para agrupar los iconos de los Workspaces, Patches, etc.

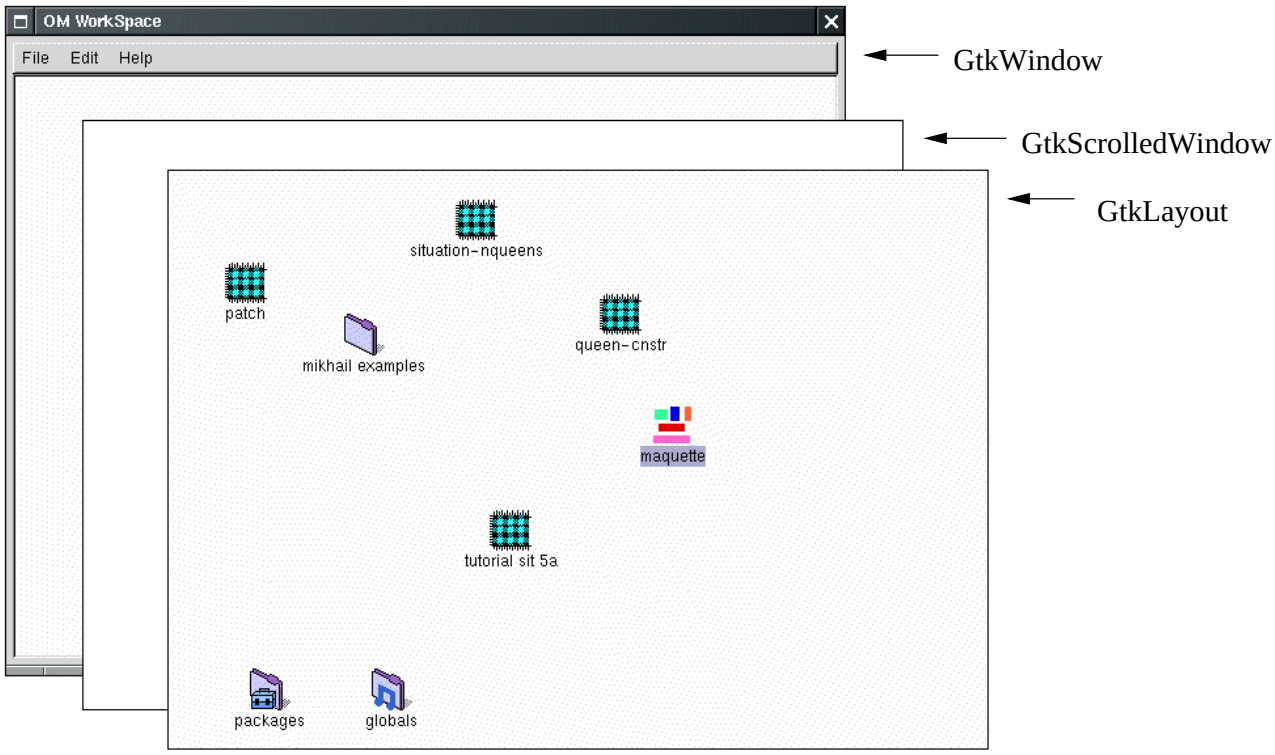

Figura 4.4: GtkWindow - GtkScrolledWindow - GtkLayout

- **Mapa de Bits (GtkPixmap)**
  Es un componente utilizado para visualizar una imagen o un icono. En la figura 4.5 se muestra un Mapa de Bits utilizado para representar un icono en OpenMusic.

- **Etiqueta (GtkLabel)**
  Es un componente utilizado para visualizar una cadena de texto. También es usado internamente por Gtk+ para visualizar texto dentro de otros componentes como los Botones (GtkButton) y los Items de Menú (GtkMenuItem). En

OpenMusic se usaron etiquetas para los nombres de los iconos como se puede apreciar en la figura 4.5.

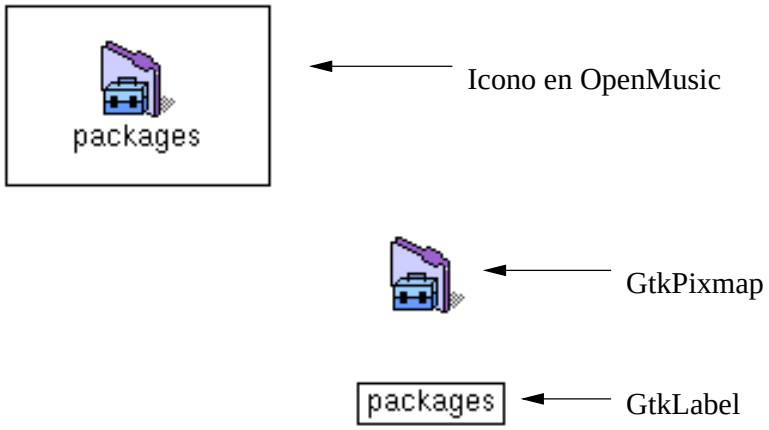

Figura 4.5: GtkPixmap - GtkLabel

- **Botón (GtkButton)**
  Es un componente utilizado para ejecutar una función específica cuando es presionado. Su presentación más simple incluye únicamente una etiqueta que describe la función del botón mismo, una presentación un poco más elaborada y en algunos casos más amigable para el usuario puede incluir una imagen que describa la función del botón. La figura 4.6 muestra una ventana donde fueron usados dos botones.

- **Entrada de Texto (GtkEntry)**
  Es un componente de entrada de texto de una sola línea, incluye una serie bastante amplia de accesos rápidos con teclado (Copiar, Pegar, etc.). El uso de la Entrada de Texto se puede apreciar en la figura 4.6, donde es usada para introducir el nombre de una función en OpenMusic.

- **Entrada de Texto Múltiple (GtkText)**
  Es un componente de entrada de texto de varias líneas, presenta las mismas características de GtkEntry. En la figura 4.6 se puede observar una Entrada de Texto Múltiple usada para introducir la documentación de una función en OpenMusic.

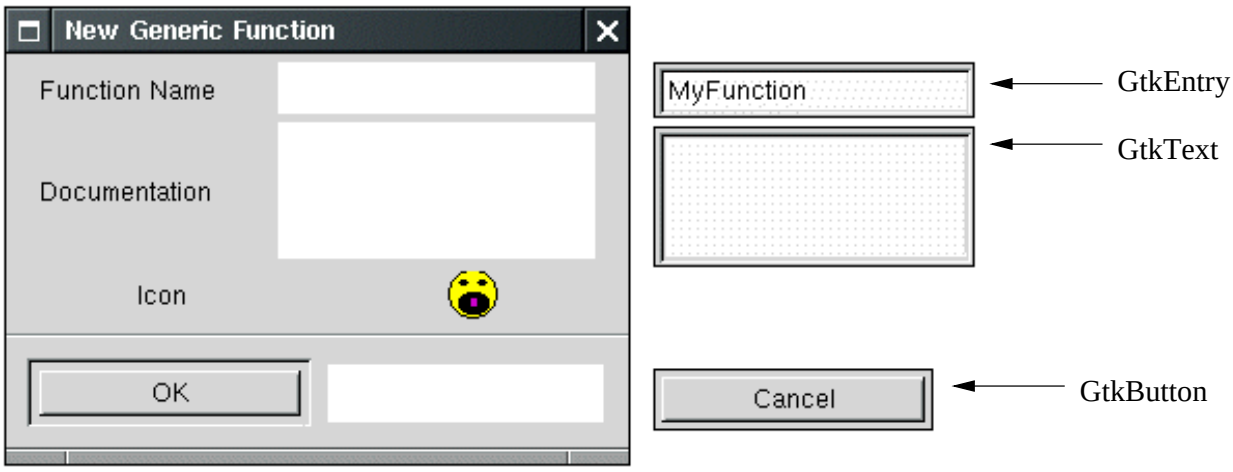

Figura 4.6: GtkEntry - GtkText - GtkButton

5 Diseño de OM Linux

El presente capítulo ilustra el diseño de los módulos considerados en la implementación del porte de OpenMusic usando CMUCL y GTK+.

5.1. Resultados de la exploración del código

La exploración del código permitio identificar cuatro bloques cuya implementación en el porte fue modularizada y llevada a cabo bajo un esquema propio para cada bloque. Estos cuatro bloques son:

- **OM ANSI**
  Esta parte del código es compatible tanto con el compilador original (MCL) como con el compilador destino (CMUCL).
- **OM NO ANSI**
  Esta parte del código se compone principalmente de funciones propias del compilador original que tienen su función equivalente en el compilador destino.
- **OM 100 % MCL**
  Esta parte del código esta compuesta por funciones propias del compilador original que no tienen una función equivalente en en compilador destino.
- **OM PARTE GRAFICA**
  Esta parte del código abarca todas las funciones que hacen uso de las clases propias del compilador original para generar interfaces gráficas de usuario.

5.2. Esquemas de implementación por bloques

Los esquemas de implementación para cada bloque se detallan a continuación (ver figura 5.1), en conjunto cada esquema permite preservar la modularidad de todo el sistema en el porte.

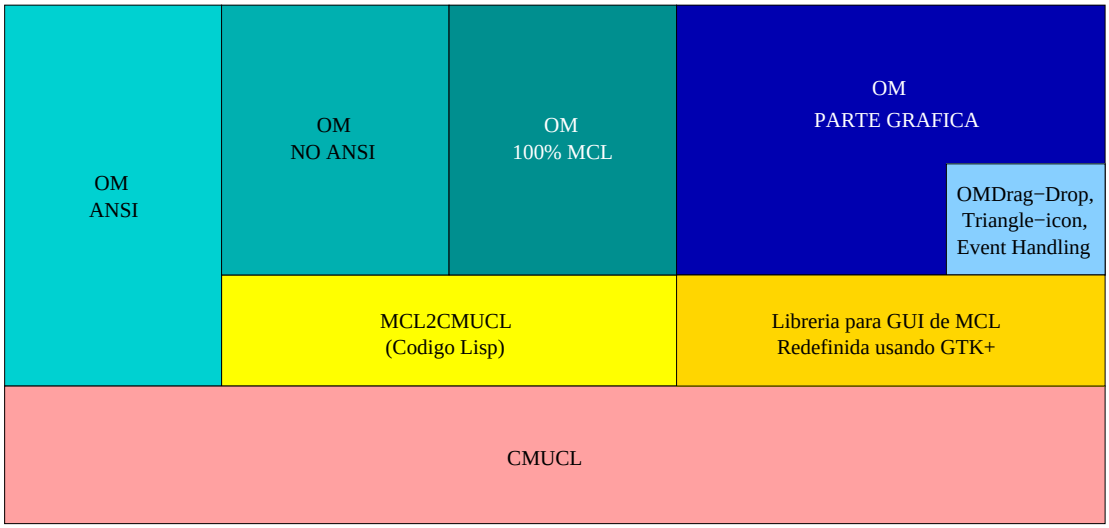

Figura 5.1: Detalle de los bloques de implementación del porte de OM

5.2.1. OM ANSI

El código que se regía estrictamente por el estandár ANSI [Ste90], se preservó en su forma original.

5.2.2. OM NO ANSI

Algunas funciones utilizadas en la implementación original de OM en MCL no se encontraban definidas en el estándar ANSI, pero existían funciones correspondientes en CMUCL que servían para el mismo propósito. Para portar este tipo de funciones se optó por un esquema de "traducción" implementado en el archivo mcl2cmucl.lisp, esto es, declarar las funciones en CMUCL con su nombre original de MCL y en el código de estas funciones invocar su función correspondiente en CMUCL.

5.2.3. OM 100 % MCL

Adicionalmente a las funciones consideradas en el bloque anterior se encontraron otras funciones no definidas en el estándar ANSI para las cuales no existía una función correspondiente en CMUCL. Este tipo de funciones debieron ser reimplementadas en CMUCL haciendo uso de los paquetes disponibles. El código para la implementación de estas funciones de incluyó también en el archivo mcl2cmucl.lisp.

5.2.4. OM Parte Gráfica

MCL cuenta con unas clases propias para la creación de interfaces gráficas, OpenMusic depende en gran parte de estas clases. GTK+ incluía todos los componentes necesarios para implementar las mismas clases de MCL y conservar el esquema original de la aplicación; el esquema de implementación fue por consiguiente reimplementar las clases de MCL para creación de interfaces gráficas usando los componentes visuales provistos por GTK+ en los archivos DialogItemsAndDialogs.lisp, Menus.lisp, ViewsAndWindows.lisp. El código para manejo de Eventos y Drag and Drop no fue reimplementado, la implantación de estos dos casos especiales se discute en el capítulo siguiente; adicionalmente el componente visual tipo arbol (usado para visualizar paquetes y las funciones que contienen de manera jerárquica), que requirío una completa implementación por parte de los desarrolladores originales de OpenMusic (MCL no incluye este tipo de componente visual, fue implementado como una clase adicional con el nombre *Triangle*), pudo ser usado directamente en la reimplementación de las clases gráficas usadas en OpenMusic ya que GTK+ incluye este tipo de componente visual.

6 Detalles de Implantación

El presente capítulo detalla las partes más significativas de la implementación del porte de OpenMusic. La primera sección cubre el rediseño obligado que debió hacerse de la arquitectura original de la aplicación. En las secciones posteriores se cubren detalles que pueden ser ubicados en los bloques identificados durante el diseño de la siguiente manera:

- **OM 100 % MCL:** Sección *Código MCL*.
- **OM Parte Gráfica:** Sección *Gráficos*.
- **OM Parte Gráfica (casos especiales):** Secciones *Eventos* y *Drag and Drop*.

Adicionalmente la sección *Implementación de Vínculos* cubre lo relacionado a las modificaciones realizadas a los vínculos de GTK+ para Common Lisp.

6.1. Clases y Metaclases

Los problemas más significativos que se tuvieron al portar OpenMusic estuvieron relacionados con las Metaclases. La metodología que se siguió en el proyecto exige respetar todo el diseño de la aplicación en el porte; sin embargo, esto no fue posible en cierta parte del código de OM debido a la flexibilidad de MCL en interpretación del Protocolo de Meta-Objetos. Aquí se enumeran los problemas en orden de aparición:

1. OpenMusic usa la clase `OMStandardClass` como metaclase de cada clase para que éstas puedan tratarse como `OMObject` y, de esta manera, pueda visualizarse y utilizarse en la programación gráfica.

El primer problema que surgió fue al declarar la clase `OMClass` como metaclase de `OMStandardClass`. La clase `OMClass` fue declarada así:

```
(defclass OMClass (pcl::standard-class OMPersistantObject)
  ((lib-class-p :initform nil :accessor lib-class-p)
   (internal-met :initform nil :accessor internal-met)))
```

es decir, `OMClass` hereda de `standard-class` (en el caso de CMUCL, `standard-class` pertenece al paquete PCL, ya que la clase `cl:standard-class` es un *wrapper*[1] encima de la verdadera clase) y de `OMPersistantObject`. Mientras que `OMStandardClass` se describió de esta manera:

```
(defclass OMStandardClass (OMClass) ()
  (:metaclass OMClass))
```

Allí surge un problema de incompatibilidad, ya que `OMStandardClass` hereda de `OMClass`, la cual, a su vez, es una instancia de `pcl::standard-class`, pero `OMStandardClass` es una instancia y también subclase de `OMClass` (ver figura 6.1). En otras palabras, las metaclases de `OMClass` (`standard-class`) y `OMStandardClass` (`OMClass`) difieren y el protocolo de meta-objetos exige que las metaclases de una clase y sus subclases sean iguales.

Para solucionar este problema se acudió a un método de validación de superclases:

```
(defmethod mop:validate-superclass ((class OMClass)
                                    (super pcl::standard-class))
  t)
```

Este método determina que la clase `pcl::standard-class` es conveniente para usarse como superclase de `OMClass`. Si el método retorna `t` para dos clases (en este caso `pcl::standard-class` y `OMClass`) con diferentes metaclases, se declara que las dos metaclases son compatibles (página 102 de [KdRB91]).

[1] *wrapper* es una clase interfaz o envoltura que cubre a otra clase.

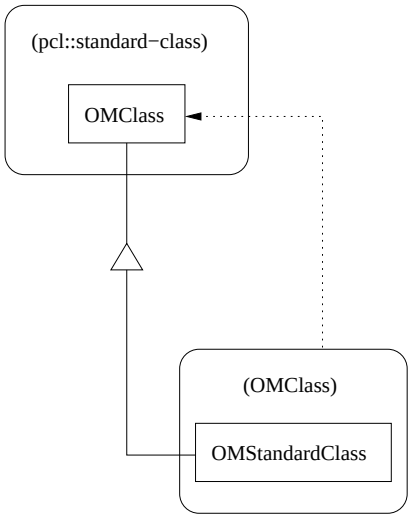

Figura 6.1: Problema de incompatibilidad de metaclases.

La razón por la cual no hay que hacer esto en MCL es que muchos compiladores no implementan el método por defecto para la validación de superclases (`mop:validate-superclass`) de una forma tan estricta como lo hace PCL.

De la solución anterior, se procedió a declarar estrictamente cada clase con `OMStandardClass` como su metaclase ya que muchas clases en su definición no tenían la línea `(:metaclass OMStandardClass)`.

2. El siguiente problema tuvo que ver con las funciones genéricas.

   En OpenMusic, las funciones genéricas son instancias de la clase `OMGenericFunction`. Esta clase tenía el siguiente código:

```
(defclass OMGenericFunction (standard-generic-function
                             OMBasicObject)
  ((numouts :initform nil :accessor numouts)
   (inputs-default :initform nil :accessor inputs-default)
   (lib-fun-p :initform nil :accessor lib-fun-p)
   (inputs-doc :initform nil :accessor inputs-doc)
   (inputs-menus :initform nil :accessor inputs-menus))
  (:default-initargs :protected-p t)
  (:metaclass OMstandardClass))
```

Los autores de OpenMusic cometieron un error al interpretar el Protocolo de Meta-Objectos [KdRB91], ya que para tener una función genérica no es suficiente heredar de `standard-generic-function`, se debe tener como metaclase a `funcallable-standard-class`[2].

Como se estaba usando `OMStandardClass` como metaclase, en compiladores estrictos como CMUCL, ésto no funciona. Además, esa es la clase de accidentes que el método `validate-superclass` intenta prevenir.

En la figura 6.2 se muestran gráficamente las clases originales.

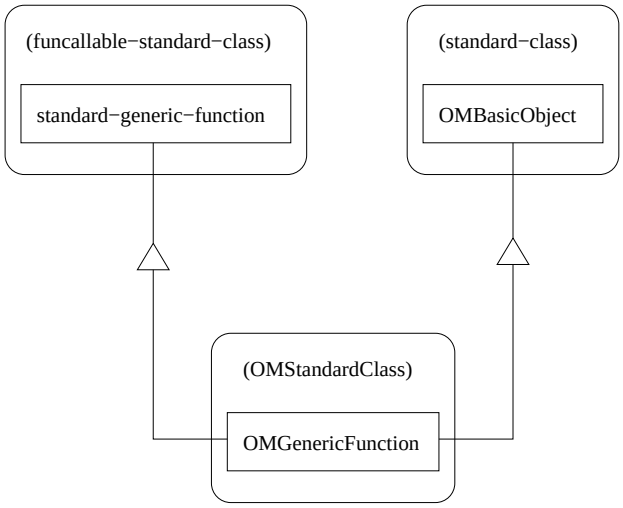

Figura 6.2: Problema de OMGenericFunction.

Para resolver este problema, en un principio se decidió crear una clase paralela a `OMStandardClass` denominada `OMFuncallableStandardClass` la cual hereda de `funcallable-standard-class` en vez de `standard-class` y redefinir `OMGenericFunction` (ver figura 6.3). Con esta solución se pretendía que `OMGenericFunction` tuviera una metaclase compatible con `funcallable-standard-class`.

```
(defclass OMGenericFunction (standard-generic-function
                             OMBasicObject)
  ((numouts :initform nil :accessor numouts)
```

[2]La metaclase de `standard-generic-function` es `funcallable-standard-class` que hace de las instancias de `standard-generic-function`, `funcallable-instances`, es decir, funciones genéricas propias.

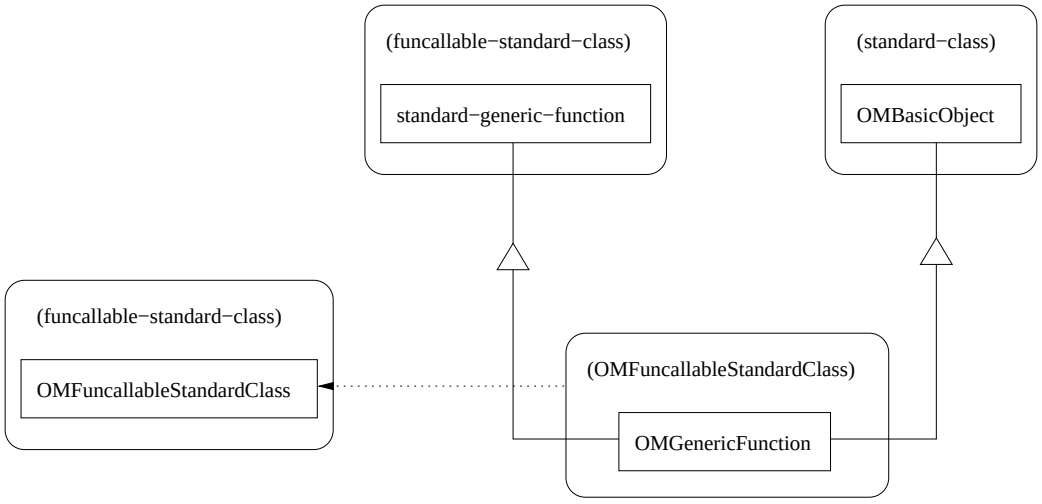

Figura 6.3: Primera solución del problema de OMGenericFunction.

```
  ...
 (inputs-menus :initform nil :accessor inputs-menus))
(:default-initargs :protected-p t)
(:metaclass OMFuncallableStandardClass))
```

sin embargo, esto no funcionó ya que había un problema de incompatibilidad con `OMBasicObject`, es decir, que las metaclases de `OMGenericFunction` (`OMFuncallableStandardClass`) y `OMBasicObject` (`standard-class`), difieren.

A raíz de esto se optó por declarar dos clases paralelas a `OMObject` y `OMBasicObject` así:

```
(defclass OMfuncallableObject (standard-generic-function)
  ((name :initform nil :initarg :name :accessor name))
  (:metaclass pcl:funcallable-standard-class))

(defclass OMfuncallableBasicObject (OMfuncallableObject)
  ((icon :initform nil :initarg :icon :accessor icon)
   (args :initform nil :initarg :args :accessor args)
   (frames :initform nil :accessor frames)
   (EditorFrame :initform nil :accessor EditorFrame)
```

```
   (attached-objs :initform nil :accessor attached-objs)
   (protected-p :initform nil :initarg :protected-p
                :accessor protected-p))
  (:metaclass pcl:funcallable-standard-class))
```

con los mismos métodos y atributos que sus correspondientes. Y luego codificar la clase de las funciones genéricas de la siguiente forma:

```
(defclass OMGenericFunction (OMfuncallableBasicObject)
  ((numouts :initform nil :accessor numouts)
   ...
   (inputs-menus :initform nil :accessor inputs-menus))
  (:default-initargs :protected-p t)
  (:metaclass pcl:funcallable-standard-class))
```

Esta solución dió resultado y las clases quedaron gráficamente como en la figura 6.4.

3. Otro problema surgió al tratar de declarar métodos en OM (los cuales se declaran con la macro `defmethod*` según el manual de desarrollo de OpenMusic [Gro98]).

   Cuando se declara un `defmethod*`, se crea una función genérica con la que puede asociarse la documentación, la clase de las funciones genéricas, la clase de los métodos, etc. (ver [Ste90]). Aquí se le encontró un error a CMUCL, que se muestra en el Anexo B.

   Luego de corregir ese error en el compilador, cuando se declaraba un `defmethod*`, el compilador se quedaba en un ciclo infinito hasta llegar a un *segmentation fault*. Se estudió detenidamente la declaración de la clase `OMMethod`, que es la clase de los métodos en las funciones genéricas, declarada como se muestra a continuación:

```
(defclass OMMethod (pcl::standard-method OMPersistantObject)
  ((saved-connections :initform nil :accessor saved-connections)
   (graph-fun :initform nil :accessor graph-fun)
   (compiled? :initform t :accessor compiled?)
```

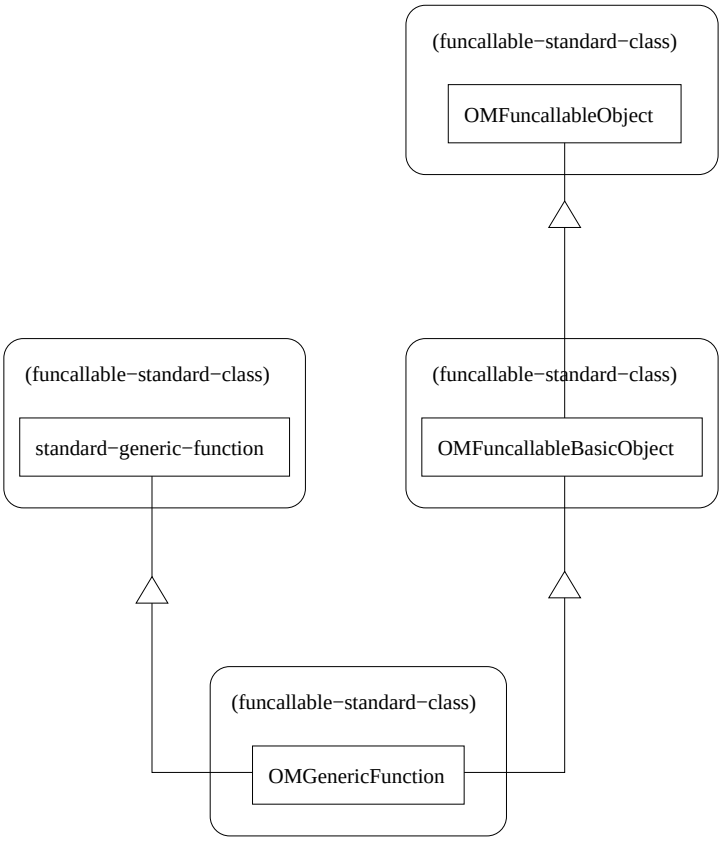

Figura 6.4: Solución final del problema de OMGenericFunction.

```
 (class-method-p :initform nil :accessor class-method-p))
(:default-initargs :protected-p t)
(:metaclass omstandardclass))
```

Se llegó a la conclusión que se debía quitar la herencia de la clase `OMPersistantObject` para remover la herencia redundante que se había formado (la cual se muestra en la figura 6.5).

Esto corrigió el error, pero fué necesario agregar a la clase, todos los atributos que `OMPersistantObject` traía y los métodos correspondientes. Gráficamente las clases quedaron como en la figura 6.6.

4. El último problema tiene que ver con las clases mismas en OM, que se declaran con la macro `defclass*`.

   Cuando se crea una clase en OpenMusic, su metaclase es `OMStandardClass`;

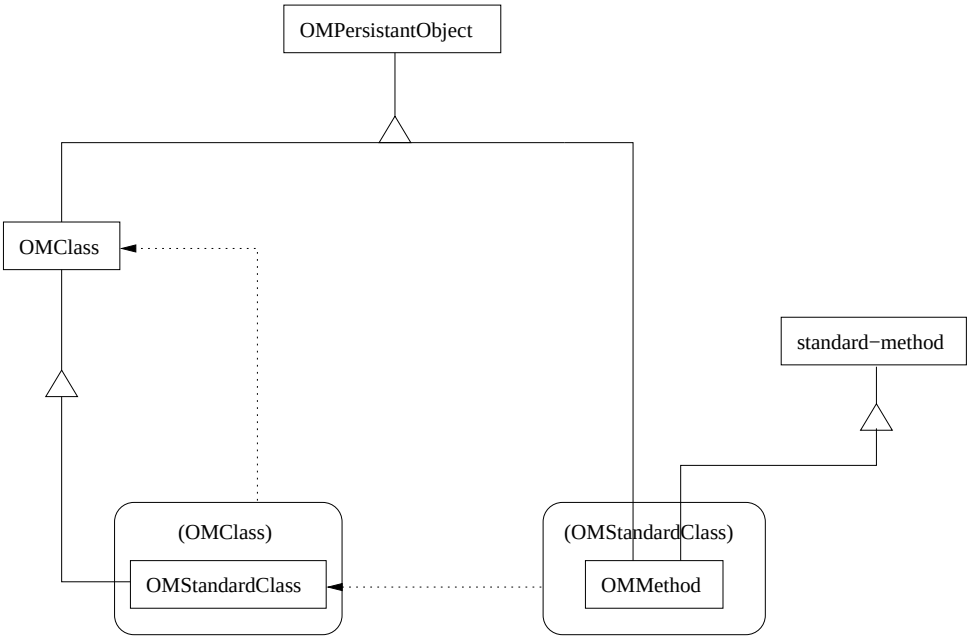

Figura 6.5: Problema de OMMethod.

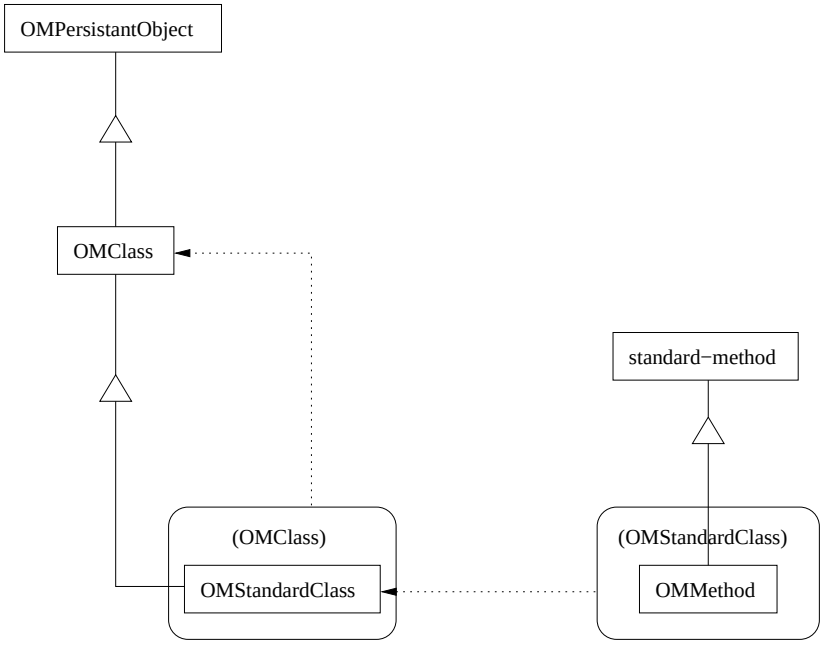

Figura 6.6: Solución del problema de OMMethod.

esto, en MCL, permite modificar sus atributos de nombre, icono, etc.

En CMUCL, éstas modificaciones no son posibles, ya que `defclass` no retorna la clase que se define, sino el *wrapper*. Es decir, si se declara la clase `uno` y luego la clase `dos`, con `uno` como su metaclase, se tiene

```
* (defclass uno (pcl::standard-class) ())
#<STANDARD-CLASS UNO {480BCCD5}>

* (defmethod mop:validate-superclass ((class uno)
                                      (super pcl::standard-class))
    t)
#<Standard-Method PCL:VALIDATE-SUPERCLASS (UNO PCL::STANDARD-CLASS)
{48C5D245}>

* (defclass dos () () (:metaclass uno))
#<STANDARD-CLASS DOS {48D5EBB5}>
```

pero el resultado de declarar la clase `dos` da como metaclase a `LISP:STANDARD-CLASS`, cuando realmente debía dar la clase `uno`:

```
#<Uno DOS {481123B5}>
```

Para que devuelva la clase real, se debe usar la función `pcl::coerce-to-pcl-class` y cuando se tiene como argumento una clase creada con `defclass*`, deben usarse las funciones propias del paquete PCL (`pcl:find-class`, `pcl:class-of`, etc.).

De allí que, para solucionar el problema, en la macro `defclass*` cuando se llama a `defclass`, se antepone el llamado a la función `pcl::coerce-to-pcl-class`. Así, si se le antepone el llamado a la función cuando se crea la clase `dos`, se tiene

```
* (pcl::coerce-to-pcl-class (defclass dos () () (:metaclass uno)))
#<Uno DOS {4802198D}>
```

6.2. Código MCL

En esta sección se muestran las implementaciones en CMUCL, de las características de MCL que se mostraron en el capítulo 3.

6.2.1. Puntos

Tal como se describió en el capítulo de Macintosh Common Lisp, un punto en MCL es la representación de una coordenada gráfica con un componente x y un componente y. Y dicho punto es codificado en un solo entero para no gastar espacio.

Los puntos son usados en OpenMusic para manipular todas los atributos gráficos, tales como posición de los iconos, de las ventanas, tamaño de las ventanas, etc.

Como CMUCL no posee la representación bidimensional de un dato en un punto, se creo un archivo denominado `MCLPoints.lisp`, donde se implementaron (con ayuda de Carlos Agon) cada una de las funciones asociadas con puntos de MCL.

También se hizo un cambio en la presentación de la macro '`#@`'. Ahora esa macro es llamada `ompoint` y se definió así:

```
(defmacro ompoint (x y)
  "This macro converts the subsequent list of two integers into a point"
  (locally (declare (fixnum x y))
           (if (and (< y (ash (1+ most-positive-fixnum) -16))
                    (>= y (ash most-negative-fixnum -16)))
               (logior (the fixnum (logand #xffff x))
                       (the fixnum (ash y 16)))
             (logior (the fixnum (logand #xffff x))
                     (ash y 16)))))
```

Luego, cada llamado del estilo `#@(10 10)` en el código de OpenMusic se reemplazó por `(ompoint 10 10)`.

La macro no fue nombrada del mismo modo que en MCL debido a que CMUCL no acepta símbolos comenzados con el caracter '`#`'.

La lista de funciones implementadas es la misma que se muestra en el capítulo 2 del Manual de Referencia de Macintosh Common Lisp [Dig96].

6.2.2. Finder Comment

Todos los archivos y directorios de Macintosh tienen una documentación que puede ser consultada sin necesidad de abrir el archivo. Puede obtenerse con la opción *general-information* del *finder* de MacOS, o, en MCL, con dos funciones: `get-finder-comment` y `set-finder-comment`.

OpenMusic usa esta documentación para guardar los datos de: posición del icono del objeto, posición de la ventana correspondiente al objeto, tamaño de dicha ventana y documentación adicional del objeto. Estos datos se crean cuando se leen los elementos del Workspace y de los folders, y se almacenan en un atributo de la clase `OMPersistantObject`, llamado `wsparams`. Este atributo luego se consulta para conocer dónde se dibuja el icono, la ventana y cual será el tamaño de ella.

Los archivos y directorios en Linux no poseen algo parecido a la documentación de Mac. Por esto, se decidió que los archivos correspondientes a objetos persistentes en OM llevaran al principio cuatro líneas de comentario de esta forma:

```
; ********************************************
; (pos-icon pos-win size-win ("Documentation"))
; type
; ********************************************
```

donde `pos-icon` es la posición del icono, `pos-win` es la posición de la ventana, `size-win` es el tamaño de la ventana y `Documentation` es la documentación adicional. Es de tener en cuenta que las posiciones y el tamaño están dados en términos de Puntos de MCL y la documentación es un string. El dato `type` es explicado en la siguiente subsección.

Para los directorios se crea un archivo dentro de ellos llamado `.finderinfo` que contiene las mismas cuatro líneas.

Se re-implementaron las funciones así:

```
(defun get-finder-comment (pathname)
  "Returns the finder comment of a file or nil if it doesn't have one"
  (let ((wsparams nil) line)
    (if (not (equal (format nil "~A" (file-namestring pathname))
                    ".finderinfo"))
        (progn
          (if (directory-pathname-p pathname)
              (setq pathname (format nil "~A/.finderinfo" pathname))
            (setq pathname (format nil "~A" pathname)))
          (with-open-file (file pathname :direction :input)
                          (read-line file)
                          (setq line (subseq (read-line file) 3))
                          (delete #\) line :count 1 :from-end t)
                          (loop for i from 0 to 2 do
                            (setq wsparams
                                  (list+ wsparams
                                    (list (read-from-string line))))
                            (setq line
                                  (remove-substring
                                    (format nil "~A" (nth i wsparams))
                                      line))))))
    wsparams))

(defun set-finder-comment (pathname comment)
  "Sets the finder comment of a file"
  (let ((data nil) line)
    (if (directory-pathname-p pathname)
        (setq pathname (format nil "~A/.finderinfo" pathname))
      (setq pathname (format nil "~A" pathname)))
    (with-open-file (file pathname :direction :input)
                    (setq line (read-line file))
                    (loop while line do
                      (push line data)
                      (setq line (read-line file nil nil))))
    (setq data (reverse data))
```

```
    (with-open-file (file pathname :direction :output
                           :if-exists :overwrite)
                    (write-line (pop data) file)
                    (write-line (format nil "; ~A" comment) file)
                    (pop data)
                    (loop while data do
                      (write-line (pop data) file)))))
```

6.2.3. Tipos de Archivos

Los usuarios del Sistema Operativo MacOS tienen al posibilidad de darle a sus archivos el tipo que quieran. OpenMusic hace uso de esta herramienta para saber qué tipo de objeto es el que están manejando con un archivo. Así, si el archivo tiene tipo `:PATC` es un Patch, `:MAQT` es una Maquette, etc.

Los archivos en linux no tiene ese tipo de diferenciación[3], por lo que adicionalmente a los datos de wsparams, se escribe el tipo en la tercera línea de los archivos creados.

La función que halla el tipo de archivo en MacOS es ahora para Linux:

```
(defun file-type (pathname)
  "Return a keyword indicating the type of pathname"
  (let (type)
    (setq pathname (format nil "~A" pathname))
    (with-open-file (file pathname :direction :input)
                    (read-line file)
                    (read-line file)
                    (setq type (read-from-string
                                (subseq (read-line file) 2))))
    type))
```

[3] Algunos manejadores de ventanas como KDE realizan la misma acción de windows para conocer el tipo: mirar la extensión del archivo. Sin embargo esto no es estándar en Linux.

6.2.4. Recursos

Adicionalmente, los archivos de MacOS tienen una serie de recursos a su disposición. Esos recursos van desde imágenes e iconos hasta fuentes.

En los recursos de la imagen de OpenMusic están los iconos de los Patches, Maquettes, etc., las fuentes que se usan, los cursores del *mouse* y la imagen que se muestra en el `OMAbout`. Estos recursos son accedidos por medio de la función `get-resources-byname` que a su vez usa la función de MCL `#_getnamedresource`.

Para contar con esta misma utilidad en Linux, se creó en el directorio `Image` un subdirectorio `.resources` donde van a estar los recursos divididos en carpetas cuyo nombre representa el tipo de recurso (Icons, Pictures, etc.). Por medio de una aplicación de MacOS, se convirtieron los iconos y la imagenes de los recursos a archivos de tipo PICT y luego en Linux se conviertieron a XPM.

```
(defun get-resources-byname (resname type)
  "Get a resources with name <resname> of type <type>"
  (let (pixmap mask bitmap)
    (if (or (equal type :pict) (equal type :icon))
        (multiple-value-bind (gdk-pixmap gdk-pixmap-mask)
            (if (equal type :pict)
                (gdk:pixmap-create
                 (string+ (printenv :omhome)
                          (format nil
                                  "/Image/.resource/Pictures/~A.xpm"
                                  resname)))
              (gdk:pixmap-create
               (string+ (printenv :omhome)
                        (format nil
                                "/Image/.resource/Icons/~A.xpm"
                                resname))))
          (setf mask gdk-pixmap-mask)
          (setf bitmap gdk-pixmap)
          (setf pixmap (pixmap-new
                        (list gdk-pixmap gdk-pixmap-mask)))))
```

```
    (values pixmap mask bitmap)))
```

De esta forma, por ejemplo, para tener el icono de los Patches, se llamaria la función con argumentos "`183`" (el nombre del archivo que contiene el Pixmap de los iconos de Patch) y "`:icon`" (el tipo de recurso).

6.2.5. Otras Funciones

OpenMusic usa, además, otras funciones que son propias de MCL para interactuar directamente con el sistema operativo. Estas funciones fueron re-implementadas en CMUCL para evitar la necesidad de alterar partes del código fuente de OpenMusic.

Dichas funciones fueron desarrolladas en el archivo `Mcl2Cmucl.lisp` y algunas de ellas usan el paquete adicional de CMUCL `unix` (el cual trae los comandos estándar de un sistema operativo tipo UNIX). Por ejemplo la función para crear un archivo usa `unix-creat`:

```
(defun create-file (pathname &key (file-type :TEXT))
  "Creates an empty file named pathname and its documentation file, and
returns the truename of the created file"
  (unix:unix-creat (format nil "~A" pathname) 436)
  (with-open-file (file (format nil "~A" pathname) :direction :output)
                  (write-line "; ************************************
**********************" file)
                  (write-line "; (0 0 0 (\"No Documentation\"))" file)
                  (write-line (format nil "; ~S" file-type) file)
                  (if (or (equalp file-type :PATC)
                          (equalp file-type :MAQT))
                      (progn
                        (write-line "; ---------------------------------
----------------------------" file)
                        (if (equalp file-type :PATC)
                            (write-line "; Patch file generated for OM
Linux" file)
                        (write-line "; Maquette file generated for
```

```
OM Linux" file))
                        (write-line "; IRCAM - Universidad Javeriana -
Cali" file)))
                  (write-line "; ************************************
**********************" file))
  (truename pathname))
```

Otro ejemplo es la función que dice si un *pathname* dado es un directorio o no; usa `unix-stat`:

```
(defun directory-pathname-p (pathname)
  "Returns a boolean value: t if pathname is a pathname specifying a
directory, nil if it is not"
  (let* ((file (format nil "~A" pathname))
         (mode (nth-value 3 (unix:unix-stat file))))
    (and (> mode 16384) (< mode 18943))))
```

6.3. Gráficos

Siguiendo la metodología propuesta, se decidió mantener la arquitectura gráfica que tiene MCL, es decir, se crearon las clases gráficas que se explicaron en la secciones de vistas, diálogos y menús del capítulo sobre Macintosh Common Lisp.

Para cada clase en OpenMusic que hereda de alguna clase gráfica, se creó un método constructor que realiza la creación de las estructuras de Gtk+ para el manejo visual. Por ejemplo, para la creación de editores se creo el siguiente constructor.

```
(defmethod initialize-instance :after ((self EditorWindow) &key)
  (let ((window (make-window :type :toplevel
                             :border-width 5
                             :visible nil)))
    (signal-connect window 'focus-in-event
     #'(lambda (event)
         (declare (ignore event))
```

```
    (setq *current-focused-window* self)))
(signal-connect window 'key-press-event
 #'(lambda (event)
    (if (not (text-view (editor (panel self))))
        (handle-key-event (panel self) (gdk:event-keyval event)
                          (gdk:event-key-state event)))
    t))
(signal-connect window 'destroy
 #'(lambda ()
    (setf (EditorFrame (object (editor self))) nil)
    (if (equalp self *om-workspace-win*)
        (progn
          (Save-before-quit)
          (om-quit)))))
(setf (widget-width window) (w self))
(setf (widget-height window) (h self))
(setf (window-title window) (om-window-title self))
(window-set-policy window 1 1 0)
(if (equal (view-position self) :centered)
    (window-set-position window :center)
  (progn
    (setf (widget-x-position window) (x self))
    (setf (widget-y-position window) (y self))))
(let ((main-box (vbox-new nil 0)))
  (container-add window main-box)
  (setf (box-spacing main-box) 0)
  (setf (container-border-width main-box) 0)
  (let ((accel-group (accel-group-new))
        (menubar (menu-bar-new)))
    (accel-group-attach accel-group window)
    (box-pack-start main-box menubar nil t 0)
    (set-gtk-menubar self menubar))
  (widget-show main-box)
  (set-main-box self main-box))
```

```
    (set-wptr self window)))
```

Dentro de la carpeta `Graphics`, se creó la carpeta `LinuxGraphics` y aquí se crearon tres archivos para cada tipo de gráfico (vistas, menús y diálogos). En cada archivo se crearon las clases correpondientes agregándole al nombre de la clase el prefijo "om-" (esto para no generar conflictos con clases de Gtk+).

Naturalmente, por ser Linux una arquitectura diferente, no se usaron la totalidad de atributos que tienen las clases en MCL, y además de eso, algunos atributos no se usaron para el mismo propósito con el cual fueron creados en Macintosh (p.ej. en la clase `simple-view`, `wptr` es el puntero a la ventana que contiene el objeto, en MCL; mientras en CMUCL, `wptr` es el puntero a la estructura Gtk+ principal del objeto). También se adicionaron otros atributos que se usaron para guardar algunas estructuras de Gtk+. En siguiente sección se muestran las clases y los atributos implementados.

6.3.1. Vistas y Ventanas

Las clases `simple-view`, `view` y `window` fueron implementadas así:

6.3.1.1. Simple-view

```
(defclass om-simple-view ()
  ((wptr :initform nil :initarg :wptr :accessor wptr)
   (main-box :initform nil :initarg :main-box :accessor main-box)
   (node :initform nil :initarg :node :accessor node)
   (view-position :initform (ompoint 0 0) :initarg :view-position
                  :accessor view-position)
   (view-size :initform (ompoint 100 100) :initarg :view-size
              :accessor view-size)
   (view-font :initform nil :initarg :view-font
              :accessor view-font)
   (selected-p :initform nil :initarg :selected-p
               :accessor selected-p)
   (help-spec :initform nil :initarg :help-spec
```

```
            :accessor help-spec)
   (view-container :initform nil :initarg :view-container
                   :accessor view-container)))
```

- wptr: El puntero a la estructura Gtk+ (i.e. Vbox, Hbox, Window, Layout)
- main-box: Primer componente hijo de la estructura instanciada en wptr en iconos o cajas (*Simple Frames*).
- node: Atributo usado sólo en la ventana de paquetes para guardar la estrucutura GtkCTreeNode.
- view-position: Al igual que en MCL, la posición del objeto.
- view-size: El tamaño del objeto.
- view-font: La fuente del objeto (no usado por ahora).
- selected-p: Si el objeto está seleccionado.
- help-spec: La ayuda del objeto (no usado).
- view-container: Contenedor donde se encuentra el objeto.

6.3.1.2. View

```
(defclass om-view (om-simple-view)
  ((scrolled-window :initform nil :initarg :scrolled-window
                    :accessor scrolled-window)
   (view-subviews :initform nil :initarg :view-subviews
                  :accessor view-subviews)))
```

- scrolled-window: Primer componente hijo de la estructura instanciada en wptr en contenedores (*Container Frames*).
- view-subviews: Subvistas del objeto.

6.3.1.3. Window

```
(defclass om-window (om-view)
  ((om-window-title :initform "Untitled" :initarg :om-window-title
                    :accessor om-window-title)
   (om-window-show :initform t :initarg :om-window-show
                   :accessor om-window-show)
   (om-window-type :initform nil :initarg :om-window-type
                   :accessor om-window-type)
   (gtk-menubar :initform nil :initarg :gtk-menubar
                :accessor gtk-menubar)
   (gtk-menubar-p :initform nil :initarg :gtk-menubar-p
                  :accessor gtk-menubar-p)
   (om-window-menu :initform nil :initarg :om-window-menu
                   :accessor om-window-menu)
   (close-box-p :initform nil :initarg :close-box-p
                :accessor close-box-p)
   (om-window-active-p :initform nil :accessor om-window-active-p)))
```

- om-window-title: Título de la ventana.
- om-window-show: Bandera que dice si la ventana se está mostrando.
- om-window-type: Tipo de ventana.
- gtk-menubar: Puntero a la barra de menús de la ventana.
- gtk-menubar-p: Bandera que dice si la barra de menús está instalada.
- om-window-menu: Menús de la ventana.
- close-box-p: Bandera que dice si la ventana tiene botón de cerrar (no usado por ahora).
- om-window-active-p: Bandera que dice si la ventana está activa.

6.3.2. Diálogos

Para el caso de los diálogos se creron las clases tal como en Macintosh, es decir, incluyendo todos sus atributos.

Las clases implementadas son:

- om-dialog-item
- om-static-text-dialog-item
- om-editable-text-dialog-item
- om-button-dialog-item
- om-check-box-dialog-item
- om-radio-button-dialog-item
- om-dialog

6.3.3. Menús

El caso de los menús es especial, ya que se declararon las clases para no tener que diseñar de nuevo toda la arquitectura de menús, pero los métodos trabajan totalmente con funciones Gtk+.

También se cambió el concepto de las variables que contienen los menús, es decir, las variables como `*patch-menubar*` que están instanciadas en la barra de menús de los Patches. Esto debido a que en MacOS, las aplicaciones pueden compartir una barra de menús, cosa que no ocurre en Linux. Para cada variable, se definió una función que crea una barra de menús nueva.

Las clases se muestran a continuación

```
(defclass om-menu-element ()
  ((menu-item-title :initform "" :initarg :menu-item-title
                    :accessor menu-item-title)
   (gtk-menu-item :initform nil :initarg :gtk-menu-item
```

```
                :accessor gtk-menu-item)
   (enabledp :initarg :enabledp :initform t
             :reader menu-item-enabled-p
             :accessor om-menu-enabled-p)
   (help-spec :initarg :help-spec :initform nil
              :accessor help-spec)))
```

- menu-item-title: Título del menú.
- gtk-menu-item: Puntero a la estructura Gtk+.
- enabledp: Bandera que dice si el menú está habilitado (no usado por ahora).
- help-spec: Ayuda del menú.

```
(defclass om-menu-item (om-menu-element)
  ((command-key)
   (menu-item-action :initform nil :initarg :menu-item-action
                     :accessor menu-item-action)))
```

- command-key: Accesso directo por teclado al menú.
- menu-item-action: Acción a realizar cuando se presiona ese menú.

```
(defclass om-menu (om-menu-element)
  ((item-list :initform nil :initarg :item-list :accessor item-list)))
```

- item-list: Lista de menú items asociados al menú.

El método más importante creado para los menús fue el siguiente:

```
(defmethod add-menu-items ((self om-menu) &rest menu-items)
  "Appends menu-items to the menu. The new items are added to the
bottom of the menu in order specified."
  (dolist (menuitem menu-items)
```

```
    (push menuitem (item-list self)))
  (if (menu-item-submenu (gtk-menu-item self))
      (setf (menu-item-submenu (gtk-menu-item self))
            (let ((menu (menu-item-submenu (gtk-menu-item self))))
              (dolist (menus menu-items)
                  (menu-append menu (gtk-menu-item menus))
                  (widget-show-all (gtk-menu-item menus)))
              menu))
    (setf (menu-item-submenu (gtk-menu-item self))
          (let ((menu (menu-new)))
            (dolist (menus menu-items)
                (menu-append menu (gtk-menu-item menus))
                (widget-show (gtk-menu-item menus)))
            menu))))
```

6.4. Eventos

Para cada objeto, OpenMusic tiene los métodos correspondientes a los eventos (p.ej. `view-click-event-handler`). Estos métodos no se usaron (a excepción de `view-key-event-handler`); en su lugar se usaron, en la constructora de cada objeto, los `signal-connect` de Gtk+. Simplemente se copió el código que existía en los métodos de MCL para cada evento.

Por ejemplo, para el caso de los `icon-box` (los iconos de las cajas), el método que se tenía era el siguiente:

```
(defmethod view-click-event-handler ((self icon-box) where)
   (cond ((command-key-p) (set-help t))
          ((double-click-p)
           (OpenObjectEditor (object (view-container self))))
         (t (toggle-icon-active-mode (view-container self))
            (when (control-key-p)
             (menu-item-context (view-container self) where)))))
```

luego, en la constructora se creó el `signal-connect`:

```
(signal-connect eventbox 'button-press-event
 #'(lambda (event)
     (setf *shift-key* (= (gdk:event-button-state event) +shift-key+))
     (when (eq (gdk:event-type event) :button-press)
       (toggle-icon-active-mode (view-container self))
       (setf *hot-point* (make-point (floor (gdk:event-x event))
                                     (floor (gdk:event-y event)))))
     (when (eq (gdk:event-type event) :2button-press)
       (OpenObjectEditor (object (view-container self))))
     t))
```

6.5. Drag and Drop

MCL no provee la herramienta de *Drag and Drop*, por lo que los creadores de OpenMusic tomaron unas funciones definidas por Dan S. Camper (lordgrey@apple.com) para el manejo de *Drag and Drop* en MCL, y le adicionaron una clase denominada `OMDrag-Drop`.

Gtk+, gracias a sus *signal-connect*, permite asignarle un evento a cualquier componente gráfico; dichos eventos incluyen los de *Drag and Drop*.

De esta forma, a cada icono de las clases gráficas que heredan de `OMSimpleFrame` (`boxframe` y `icon-finder`), en su función constructora, le fue agregado un evento *drag-begin* de la siguiente forma:

```
(signal-connect eventbox 'drag-begin
 #'(lambda (context)
     (declare (ignore context))
     (setq *OM-drag&drop-handler*
           (get-actives (panel (front-window))))
     (setq *OM-drag&drop-handler*
           (remove (get-drag-object self) *OM-drag&drop-handler*))
     (push (get-drag-object self) *OM-drag&drop-handler*)))
```

Dicho evento fue asociado al `eventbox` que contiene la imagen del icono para que así,

solamente se dispare el evento cuando se haga "click" en el dibujo y no en el nombre asociado a él. La variable `*OM-drag&drop-handler*` es la variable que guarda las instancias de las clases que están siendo arrastradas y, de esta manera, copiarlas o moverlas en el contenedor indicado por el usuario.

Por otro lado, en las funciones constructoras de las clases de los contenedores (`metaobj-panel` y `PatchPanel`) se agregó el evento *drag-drop* asociándolo al panel (`layout`) de la ventana:

```
(signal-connect layout 'drag-drop
 #'(lambda (context x y time)
     (declare (ignore time))
     (let* ((flag nil)
            (boxframe (widget-get-parent
                       (widget-get-parent
                        (drag-get-source-widget context))))
            (eventbox (drag-get-source-widget context))
            (iconwidth (widget-get-width eventbox))
            (boxwidth (widget-get-width boxframe))
           xput yput position-hard position-soft dx dy)
       (dolist (su (view-subviews self))
         (if (equalp (wptr su) boxframe)
             (setq flag t)))
       (if flag
           (progn
            (setq position-hard
                  (view-position (car *OM-drag&drop-handler*)))
            (dolist (handler *OM-drag&drop-handler*)
             (setq position-soft (view-position handler))
             (setq dx (- (point-h position-soft)
                         (point-h position-hard)))
             (setq dy (- (point-v position-soft)
                         (point-v position-hard)))
             (if (or (< (point-h *hot-point*)
                        (floor (/ iconwidth 2)))
```

```
                    (= (point-h *hot-point*)
                       (floor (/ iconwidth 2))))
                (setf xput (- x (- (floor (/ boxwidth 2))
                                   (- (floor (/ iconwidth 2))
                                      (point-h *hot-point*)))))
                (setf xput (- x (+ (floor (/ boxwidth 2))
                                    (- (point-h *hot-point*)
                                       (floor (/ iconwidth 2)))))))
                (setf yput (- y (point-v *hot-point*)))
              (OMGMoveObject handler
                             (make-point (+ dx xput) (+ dy yput)))))
            (dolist (handler *OM-drag&drop-handler*)
              (perform-drop handler self (make-point x y)))))
      (setq *OM-drag&drop-handler* nil)
      nil))
```

En este evento se declara una variable booleana `flag`, que toma el valor de `t` si los objectos del *Drag and Drop* se están moviendo en la misma ventana, o `nil` si las ventanas origen y destino son diferentes. Para esto se hace una comparación con la lista de subvistas del contenedor en busca del mismo objeto que se está soltando en el contenedor. Si el objeto está en las subvistas del contenedor (`flag` es `t`) simplemente se hace un desplazamiento del componente en el panel y se actualiza la posición del objeto; de lo contrario (`flag` es `nil`) se agrega el elemento al nuevo contenedor y, dependiendo de si se está copiando o se está moviendo, se borra del contenedor fuente.

6.6. Implementación de Vínculos

El uso de GTK+ en Common Lisp fue posible gracias a una implementación en Lisp de los vínculos (*bindings*) de la librería en C escrita por Espen S. Johnsen. Esta implementación incluye el manejo de los tipos básicos de GTK y GLib, así como la mayoría de las funciones para la manipulación de los componentes visuales más comúnes.

Inicialmente no fue necesario modificar el código de los vínculos puesto que las primeras tareas para el porte incluían la creación de ventanas e iconos, para lo cual se encontraban disponibles todas las funciones necesarias en Lisp; más adelante empezaron a ser necesarias funciones para dibujar líneas y polígonos en GTK+, que no se encontraban disponibles en Lisp, lo que hizo necesario modificar el código de los vínculos para incluirlas. Este tipo de modificación inicial se lleva a cabo de la siguiente manera:

- El prototipo de la función en C es el punto de partida para poder escribir el vínculo, por ejemplo, la siguiente función para dibujar lineas:

```
void gdk_draw_line(GdkDrawable *drawable,
                   GdkGC *gc,
                   gint x1,
                   gint y1,
                   gint x2,
                   gint y2);
```

- Se procede a hacer el chequeo de los tipos de los argumentos de la función, esto es, verificar si todos los tipos usados en la función se encuentran declarados en Lisp. Para esto se consulta la documentación de la librería en C que permite establecer para cada tipo si pertenece a GTK o a GDK, acto seguido se debe consultar el código fuente correspondiente en Lisp: gtktypes.lisp para los tipos de GTK, y gdktypes.lisp para los tipos de GDK. Para el caso del porte del kernel no fue necesario definir ningún tipo adicional de GTK o GDK, puesto que los vínculos incluían todos los tipos necesarios para la implementación.

- A continuación se debe definir la función en C como una función foránea en Lisp; para esto se usa la función `define-foreign`. Para el caso del ejemplo, se definiría de la siguiente forma (dentro del archivo correspondiente, bien sea gtk.lisp o gdk.lisp):

```
(define-foreign draw-line () none
                (drawable (or window pixmap bitmap))
                (gc gc)
                (x1 int)
```

```
(y1 int)
(x2 int)
(y2 int))
```

donde `draw-line` es el nombre de la función en lisp (que corresponde al nombre de la función en C sin el prefijo gtk o gdk y reemplazando el caracter '_' por '-'), `none` es el tipo del dato que retorna la función (`none` corresponde al `void` de C), y `drawable`, `gc`, `x1`, `y1`, `x2` y `y2` son los parámetros de la función en C, con sus respectivos tipos en Lisp.

- Finalmente se deben recompilar los vínculos y se puede hacer uso de la función en Lisp con el nombre asignado (`draw-line` en el caso del ejemplo) y los parámetros requeridos.

Posteriormente fue necesario implementar funciones especiales para acceder internamente a estructuras en C. Este tipo de implementación se describe a continuación:

- Como primera instancia se deben identificar la estructura en C y los campos de la misma que se desean manipular en Lisp. Tómese como ejemplo la siguiente estructura en C:

```
struct GtkRequisition{
                      gint16 width;
                      gint16 height;
                     };
```

- En la estructura anterior se desea acceder a ambos campos (`width` y `height`), para lo cual se implementan dos funciones en C que posteriormente serán declaradas como funciones foráneas en Lisp. Las funciones en C son:

```
gint gtk_widget_get_width (GtkWidget *widget)
{
  GtkRequisition req;
  gtk_widget_size_request(widget, &req);
  return req.width;
}
```

```
gint gtk_widget_get_height (GtkWidget *widget)
{
  GtkRequisition req;
  gtk_widget_size_request(widget, &req);
  return req.height;
}
```

Cabe destacar que en el caso del ejemplo los campos de las estructuras se accesan una vez que han sido modificados al invocar la función `gtk_widget_size_request`.

- El código en C de estas funciones debe ser incluido en el archivo `cl-gtk.c` que hace parte de los vínculos.
- Se procede a definir las funciones foráneas en Lisp. Dado que las funciones del ejemplo fueron definidas con con el prefijo `gtk` se deben declarar en el archivo gtk.lisp como se describe a continuación:

```
(define-foreign widget-get-width () int
                (widget widget))

(define-foreign widget-get-height () int
                (widget widget))
```

Las funciones anteriores retornan un valor de tipo `int` y reciben como único parámetro una variable de tipo `widget`, los dos tipos se encuentran definidos en Lisp.

- Se recompilan nuevamente los vínculos para poder hacer uso de las funciones en Lisp.

Como última modificación para los vínculos se implementaron funciones especiales en C encargadas de manipular internamente estructuras de GTK y GDK para ser usadas en llamados a otras funciones, en estos casos no se retorna el valor de ninguno de los campos de la estructura. La implementación de este tipo de funciones es bastante similar a las del tipo anterior.

7 Conclusiones

- Es posible implementar eficientemente OpenMusic en Linux, a pesar de estar concebido originalmente con base en las herramientas gráficas de MacOS y MCL.

- El diseño de OpenMusic no sigue los patrones del protocolo de meta-objetos, cosa que es muy necesaria en el desarrollo de una aplicación en Common Lisp y CLOS, para evitar posteriores rediseños.

- Se definió una metodología para el porte de de aplicaciones en Common Lisp, ya que el porte de un programa, visto como proyecto de desarrollo de software, requiere de una. Las particularidades del porte de OpenMusic y en general de cualquier otro, hacen que la metodología a seguir pueda variar dependiendo de los factores adicionales que se deban considerar, destacandose entre éstos las plataformas escogidas (origen y destino), el o los lenguajes de implementación involucrados (el lenguaje de origen puede ser distinto al de destino), etc.

- La escogencia de las herramientas de desarrollo para un porte está condicionada por varios factores, entre los que se debe contar, de manera principal, el de cumplir con los objetivos trazados al principio del proyecto en el tiempo estipulado para cada uno. Para el caso discutido en este documento se puede considerar que de haber usado una herramienta de desarrollo para la interfaz gráfica que no incluyera un conjunto de componentes visuales tan amplio como el de GTK+ hubiera sido necesario diseñar y desarrollar el conjunto de componentes requerido por la aplicación, labor que hubiera alterado y prolongado el desarrollo del proyecto.

- El desarrollo con herramientas de uso público (p.ej. herramientas GNU) permite acceder a un soporte que no difiere del de las herramientas comerciales,

por el contrario puede contarse con soporte directo por parte de los autores mismos de la herramienta (quien escribió el código), además de disponer siempre del código fuente de la misma y de los grupos de desarrolladores alrededor del mundo que se comunican mediante listas de correo.

- Cuando se sigue un proceso de desarrollo formal para un proyecto de software los resultados están soportados por una base formal, lo cual posibilita que proyectos futuros (p.ej. portes) en base al proyecto inicial puedan ser desarrollados de manera formal y sin requerir modificaciones en el diseño del proyecto original. En el desarrollo de este proyecto no se contó con un documento de especificación formal de la aplicación original que hubiera servido para desarrollar un especificación formal del porte.

- El API del sistema operativo MacOS provee una interfaz gráfica nativa para cualquier implementación que supera en rapidez a la interfaz gráfica provista por GTK+ para desarrollo de aplicaciones en Linux.

- MCL provee al programador de un ambiente integrado de desarrollo que incluye una librería para interfaces gráficas y manejo de eventos, razón por la cual no es requerido un desarrollo por componentes (usando varias herramientas). Por el contrario en Linux el mismo tipo de desarrollo fue posible usando varias herramientas (CMUCL, GTK+, vínculos de GTK+ para Lisp).

8 Recomendaciones

- Se deben desarrollar los editores adicionales de OpenMusic (Maquettes, Bpf, editores musicales, etc.) para proveer a los usuarios especializados de OM (en especial a los músicos) de todas las herramientas que tiene OM, ya que ahí radica el poder de la aplicación.
- Incorporar a OpenMusic bajo Macintosh, las opciones para exportar Patches a OM Linux e importar Patches desde OM Linux.
- Desarrollar un entorno de edición de programas en Lisp, con evaluación incremental.
- Explorar la conveniencia de integrar editores musicales como Rosegarden.
- Analizar la posibilidad de integrar *Midishare* bajo Linux en OpenMusic (utilizando la reciente propuesta de Yann Oraley) y utilizarla para la generación de sonido.
- Integrar utilidades para la lectura y escritura de archivos MIDI, WAV, AIFF, etc.
- Al haber desarrollado la versión para Linux de OpenMusic en la Universidad Javeriana es indispensable que, para futuros desarrollos conjuntos con el IRCAM sobre la misma aplicación u otros proyectos, se monte un servidor de control de versiones (CVS).
- Es importante seguir desarrollando trabajos de grado de manera conjunta con universidades e institutos internacionales para tener una difusión más amplia y un desarrollo continuo de los mismos.

Bibliografía

[AC92] Inc. Apple Computer. *Inside Macintosh: Macintosh ToolBox Essentials*. Apple Computer, Inc., 1992.

[AC93] Inc. Apple Computer. *Inside Macintosh: More Macintosh ToolBox*. Apple Computer, Inc., 1993.

[Dig96] Inc. Digitool. *Macintosh Common Lisp Reference*. Digitool, Inc., 1996.

[GM01] Tony Gale and Ian Main. *Gtk 1.2 Tutorial*. http://www.gtk.org/tutorial, March 2001.

[Gro98] Ircam Users Group. *OpenMusic Developer's Documentation*, 1998.

[Gro99] Ircam Users Group. *OpenMusic User's Manual and Tutorial*, 1999.

[KdRB91] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*, chapter 5 y 6. MIT Press, 1991.

[Kee89] Sonya E. Keene. *Object-Oriented Programming in Common Lisp*. Symbolics, Inc., 1989.

[Mac01] Robert A. MacLachlan. *CMU Common Lisp User's Manual*. http://cvs2.cons.org:8000/cmucl/doc/cmu-user/, 2001.

[Pit96] Kent Pitman. *Common Lisp HyperSpec (TM)*. 1996.

[Pre88] Roger S. Pressman. *Ingeniería del Software*. McGraw-Hill, 1988.

[RAAL99] Camilo Rueda, Carlos Agon, Gerard Assayag, and Mickael Laurson. *Computer assisted composition at Ircam: Patch Work and OpenMusic*. 1999.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Loransen. *Object-Oriented Modeling and Design*. Prentice Hall, Inc., 1991.

[Ste90] Guy L. Steele. *Common Lisp the Language, 2nd Edition*. Digital Press, 1990.

[Tea01] Gtk+ Team. *Gtk+ API Reference*. http://www.gtk.org/api, 2001.

Anexo A. Manejo de OpenMusic en Linux

Anexo A. Manejo de OpenMusic en Linux

REQUERIMIENTOS

- CMUCL 2.4.22 Instalado.
- GTK+ 1.2.x Instalada.

IMAGEN

Para hacer uso de OpenMusic en Linux, al igual que en Macintosh, se debe compilar la imagen. Para esto, se debe descomprimir el archivo de la distribución de OpenMusic, ir a la carpeta `Build-Image` y ejecutar '`make`'. Lo anterior dejará la imagen en la carpeta `Image`, a continuación se podrá ejecutar el script '`omlinux`' (en la carpeta `Image`) para arrancar la aplicación.

USO

OpenMusic para Linux tiene las mismas características que su similar en Macintosh, y su uso es igual (el cual se puede ver en el Manual de Usuario y Tutorial de OpenMusic [Gro99]) excepto por las siguientes diferencias:

- El Workspace y sus elementos, las variables globales, las preferencias y las librerías de usuario se guardan en un directorio denominado *.openmusic* que se crea en el *home* del usuario que ejecuta la aplicación.
- Para sacar un cuadro de texto para entrar funciones de OM o Lisp, se debe arrastrar el *mouse* con la tecla CTRL (en lugar de la tecla APPLE en Macintosh) y el botón izquierdo del mouse presionados.
- Los menús contextuales se obtienen haciendo "click" con el botón derecho.
- Para evaluar un objeto, se debe hacer "click" con la tecla WINDOW presionada.

- Para agregar entradas a los objetos se debe pulsar la tecla FLECHA-DERECHA con la tecla WINDOW presionada. Para remover entradas se pulsa FLECHA-IZQUIERDA con la tecla WINDOW presionada.

- Los Patches que generados en la versión de OpenMusic para Macintosh (que no contengan clases musicales, ya que éstas se encuentran fuera del alcance del trabajo de grado) pueden cargarse en linux, copiando el archivo correspondiente a
  .openmusic/Workspaces/OM Workspace/elements/
  y agregándole las siguientes 7 líneas al principio:

```
; ************************************************************
; (0 0 0 ("no documentation"))
; :PATC
; ------------------------------------------------------------
; Patch file generated for OM Linux
; IRCAM - Universidad Javeriana - Cali
; ************************************************************
```

  Las líneas 2 y 3 son las más importantes y deben copiarse de forma exacta (incluso el espacio entre el punto y coma, y el primer paréntesis).
  Las otras líneas pueden ser reemplazadas, pero deben existir (como comentarios en lisp, esto es, empezando con punto y coma) ya que OpenMusic lee estas líneas al carga el Workspace y las escribe al guardarlo.

- Las librerías de usuario deben ubicarse en *.openmusic/User Library*.

Anexo C. Metodología

Anexo C. Metodología

En la actualidad existen metodologías para desarrollo de software desde su concepción tales como cascada y espiral. La ingeniería de software nos guía en la escogencia del modelo más adecuado para cada desarrollo en particular, dependiendo de las ventajas y desventajas que presente la aplicación de cada uno de los modelos en el proyecto.

Para el caso particular de un proyecto que pretenda portar un programa entre dos plataformas, no existe una metodología definida. Por esta razón, al inicio de este proyecto, se diseño una metodología para el porte de programas en Lisp cuando el código involucra interfaces gráficas o uso de librerías nativas del compilador de origen que no cumplan con el estándar ANSI Common Lisp.

La metodología diseñada se compone de seis etapas:

1. Familiarización con la aplicación.
2. Exploración del código.
3. Escogencia de las herramientas de desarrollo.
4. Diseño de los módulos (objetos, funciones, GUI's, etc.) en la herramienta escogida.
5. Implantación y pruebas en la nueva plataforma.

Haciendo una analogía con las cuatro etapas de un modelo de desarrollo clásico, las tres primeras etapas de nuestra metodología podrían considerarse como el análisis del proyecto y las otras tres como el diseño, implementación y pruebas, respectivamente.

FAMILIARIZACIÓN CON LA APLICACIÓN

En esta primera etapa, se debe llevar a cabo la compilación e instalación típica de la aplicación en la plataforma origen.

Se ejecuta el programa para explorar todas sus características: forma de interacción con el usuario, disposición de la interfaz gráfica (menús, área de trabajo, área de edición, áreas de entrada y salida de datos, eventos que maneja la aplicación, etc.). La exploración del programa en sí, debe ir más allá de lo que requiere un usuario de la aplicación y debe llegar al punto de identificar gráficamente los módulos más primitivos de la aplicación. El código correspondiente se identifica en la siguiente etapa.

EXPLORACIÓN DEL CÓDIGO

Aquí se deben identificar y comprender las estructuras Common Lisp y CLOS usadas dentro del código de la aplicación, la jerarquía de módulos y las librerías usadas para generar la interfaz gráfica.

Debe hacerse un seguimiento minucioso de los llamados de las funciones, empezando por idenficar el módulo principal e ir paso a paso en la ejecución del programa para que de esta forma pueda verse claramente cuales son las funciones, clases, métodos y variables que componen la arquitectura de la aplicación.

Por ejemplo, en OpenMusic se debe identificar que el archivo más importante es `modele.lisp`, ya que en éste se encuentra la definición de la clase `OMObject` y sus subclases principales; se debe iniciar el seguimiento por el archivo `image-init.lisp`: aquí se llama a la función `show-workspaces-dialog` (que se encuentra en el archivo `dialogs.lisp`), ésta crea los directorios en el *home* del usuario, y llama a `init-remote/local` (en el archivo `OMWorkSpace.lisp`) que a su vez llama a `init-OM-sesion`, etc.

ESCOGENCIA DE HERRAMIENTAS DE DESARROLLO

La escogencia del compilador debe empezar por una búsqueda profunda que arroje el conjunto de compiladores disponibles en la plataforma destino para Common

Lisp y si la aplicación tiene módulos con programación orientada a objetos, dichos compiladores deben proveer además una implementación completa de CLOS.

Las ventajas y desventajas de cada uno de los compiladores encontrados pueden ser identificadas mediante aplicaciones ya desarrolladas en los mismos, pruebas con algoritmos propios de los desarrolladores que harán el porte y mediante la opinión de otros desarrolladores que pueden ser contactados mediante listas de correo.

De allí, el conjunto de compiladores candidatos se reduce; y se debe proceder a probar, con cada uno de los compiladores, que éstos tengan soporte para las características del código que se encontraron en la etapa anterior. El nuevo conjunto reducido de compiladores no implica descartar los otros compiladores, puesto que aún deben llevarse a cabo las pruebas con las librerías para interfaz gráfica.

El paso siguiente es el de la escogencia de la librería para generar la interfaz gráfica de la aplicación. Generalmente en el manual de referencia de cada compilador se incluye una lista de las librerías para interfaz gráfica admitidas por el mismo; por defecto, todos los compiladores de Common Lisp en Unix incluyen el paquete CLX correspondiente a la librería XLib (perteneciente al protocolo X), que debe ser usada sólo cuando se requiera un manejo gráfico a muy bajo nivel. En el caso contrario (manejo gráfico de alto nivel con un conjunto amplio de componentes y eventos), se debe escoger una librería que ofrezca una gran semejanza con la librería usada en el compilador de origen.

La combinación de los dos pasos anteriores debe arrojar el compilador y la librería para interfaz gráfica más adecuados para el porte.

DISEÑO DE LOS MÓDULOS

Teniendo claro cuál es el compilador a usar, sabiendo cuáles son los paquetes que éste tiene, conociendo la librería gráfica y habiendo explorado el código de origen, se procede ahora a buscar aquellas funciones, métodos y clases que no podrán ser portados directamente y que deberán ser reemplazados por sus correspondientes en el compilador destino.

Para cada función que no se encuentre en un paquete del compilador destino, debe buscarse en la documentación del compilador origen sus entradas, salidas y com-

portamiento. Luego se implementan las funciones en un archivo nuevo, creado para dicho fin, con sus mismos nombres e incluídos en el paquete donde se encuentre la aplicación. Esto con el fin de no alterar, de una forma sustancial, el código original.

Para la parte relacionada con la interfaz gráfica se debe optar por no alterar la arquitectura definida en la aplicación original, puesto que cualquier cambio puede obligar a reescribir gran parte del código que depende de la interfaz gráfica o incluso rediseñar la arquitectura propia de la aplicación (lo cual es un gran error, ya que en un porte la aplicación no debe ser rediseñada). El camino adecuado debe respetar la arquitectura gráfica y planear el reemplazo de las funciones gráficas una por una, siendo necesario en algunos casos la construcción de funciones especiales, que suelen estar incluidas en la librería original, pero que no tienen su correspondiente (o semejante) en la librería destino.

IMPLANTACIÓN EN LA NUEVA PLATAFORMA

Siguiendo el camino trazado durante la etapa anterior, se procede a implantar en orden de jerarquía cada uno de los módulos en el compilador escogido, junto con las interfaces gráficas a partir de la librería escogida.

La parte de pruebas va muy ligada a la implantación dentro de la metodología definida, se verifica el correcto funcionamiento de cada una de las características de la aplicación y se comprueba que el rendimiento de la misma sea semejante o mejor al del programa original.

La labor de probar el porte de una aplicación difiere de la de probar una aplicación desarrollada desde cero; la diferencia radica en que el porte debe ser probado adicionalmente contra la aplicación original para constatar que su funcionalidad, manejo y apariencia son muy similares.

Anexo C. Comunicación con Pierre R. Mai, vía E-mail

Anexo C. Comunicación con Pierre R. Mai, vía E-mail

```
Date: Thu, 10 May 2001 11:27:23 -0500 (COT)
From: Gerardo M. Sarria M. <gsarria@escher.puj.edu.co>
To: cmucl-help <cmucl-help@cons.org>
Subject: defgeneric and :method-class

Hello!

When I define a generic-function with defgeneric, and I set the
:method-class with another class that inherits from pcl::standard-method,
it shows an error.

I mean, I define a class

     (defclass themethod (pcl::standard-method) ())

Then I create a generic-function

     (defgeneric foo (x y)
          (:method-class themethod))

The error:

Error in function PCL::|(FAST-METHOD NO-APPLICABLE-METHOD (T))|:
   No matching method for the generic-function #<Standard-Generic-Function
PCL:CLASS-PROTOTYPE (4) {281C45B9}>,
when called with arguments (THEMETHOD).

Restarts:
  0: [CONTINUE] Retry call to #<Standard-Generic-Function
PCL:CLASS-PROTOTYPE (4) {281C45B9}>
  1: [ABORT   ] Return to Top-Levell.
```

```
Debug  (type H for help)

(PCL::|(FAST-METHOD NO-APPLICABLE-METHOD (T))| #<unused-arg>
                                               #<unused-arg>
                                               #<Standard-Generic-Function
PCL:CLASS-PROTOTYPE (4) {281C45B9}>
                                               (THEMETHOD))
Source: Error finding source:
Error in function DEBUG::GET-FILE-TOP-LEVEL-FORM:  Source file no longer
exists:
  target:pcl/braid.lisp.
```

Can somebody told me what I did wrong and how can I solve this?

Thanks

Gerardo M.

Date: 10 May 2001 19:02:11 +0200
From: Pierre R. Mai <pmai@acm.org>
To: cmucl-help@cons.org
Cc: Gerardo M. Sarria M. <gsarria@escher.puj.edu.co>
Subject: Re: defgeneric and :method-class

"Gerardo M. Sarria M." <gsarria@escher.puj.edu.co> writes:

> Hello!
>
> When I define a generic-function with defgeneric, and I set the
> :method-class with another class that inherits from pcl::standard-method,

```
> it shows an error.
>
> I mean, I define a class
>
>      (defclass themethod (pcl::standard-method) ())
>
> Then I create a generic-function
>
>      (defgeneric foo (x y)
>           (:method-class themethod))
>
> The error:
>
> Error in function PCL::|(FAST-METHOD NO-APPLICABLE-METHOD (T))|:
>    No matching method for the generic-function #<Standard-Generic-Function
> PCL:CLASS-PROTOTYPE (4) {281C45B9}>,
> when called with arguments (THEMETHOD).
>
> Restarts:
>   0: [CONTINUE] Retry call to #<Standard-Generic-Function
> PCL:CLASS-PROTOTYPE (4) {281C45B9}>
>   1: [ABORT   ] Return to Top-Levell.
>
> Debug  (type H for help)
>
> (PCL::|(FAST-METHOD NO-APPLICABLE-METHOD (T))| #<unused-arg>
>                                                #<unused-arg>
>                                                #<Standard-Generic-Function
> PCL:CLASS-PROTOTYPE (4) {281C45B9}>
>                                                (THEMETHOD))
> Source: Error finding source:
> Error in function DEBUG::GET-FILE-TOP-LEVEL-FORM:  Source file no longer
> exists:
>   target:pcl/braid.lisp.
```

>

It seems that the following method definition is missing from PCL:

```
(defmethod pcl:class-prototype ((class symbol))
  (pcl:class-prototype (pcl::find-class class)))
```

OTOH maybe this is an error in the defgeneric macro processing, which should probably use find-class on the :method-class parameter before passing it to class-prototype. I'll investigate and produce a fix. In the meantime the given method definition seems to work...

Regs, Pierre.

--
Pierre R. Mai <pmai@acm.org> http://www.pmsf.de/pmai/
The most likely way for the world to be destroyed, most experts agree, is by accident. That's where we come in; we're computer professionals. We cause accidents. -- Nathaniel Borenstein

---

Date: 10 May 2001 21:08:38 +0200
From: Pierre R. Mai <pmai@acm.org>
To: cmucl-help@cons.org
Cc: Gerardo M. Sarria M. <gsarria@escher.puj.edu.co>
Subject: Re: defgeneric and :method-class

"Pierre R. Mai" <pmai@acm.org> writes:

> "Gerardo M. Sarria M." <gsarria@escher.puj.edu.co> writes:
>
> > Hello!
> >
> > When I define a generic-function with defgeneric, and I set the

> > :method-class with another class that inherits from pcl::standard-method,
> > it shows an error.

[...]

> It seems that the following method definition is missing from PCL:
>
> (defmethod pcl:class-prototype ((class symbol))
>   (pcl:class-prototype (pcl::find-class class)))
>
> OTOH maybe this is an error in the defgeneric macro processing, which
> should probably use find-class on the :method-class parameter before
> passing it to class-prototype. I'll investigate and produce a fix.
> In the meantime the given method definition seems to work...

It is indeed an omission in the argument processing done by ensure-generic-function-using-class, which fails to coerce the :method-class argument to a class object, as specified in the AMOP, page 187 (entry for ensure-generic-function-using-class).

Hence generic-function-method-class will return the name and not the class all over the place, and this could in theory lead to other errors, beside the one Gerardo encountered. A correct fix is about to be comitted, but this fix is non-trivial to install in a running image, hence the defmethod stuff might be used as a temporary work-around, until new binaries are released (I'll probably compile up new binaries for x86/Linux in the next couple of weeks).

If someone really needs this fixed in his 18c binary, I could produce a file that does the correct fixes at load-time. Contact me in that case.

Regs, Pierre.

--
Pierre R. Mai <pmai@acm.org> http://www.pmsf.de/pmai/
The most likely way for the world to be destroyed, most experts agree,

---

Date: 12 May 2001 19:16:21 +0200
From: Pierre R. Mai <pmai@acm.org>
To: GerardoM SarriaM <gmsarria@tutopia.com>
Cc: gsarria@escher.puj.edu.co
Subject: Re: defgeneric and :method-class
Parts/Attachments:
1 Shown 47 lines Text
2 3.8 KB Application
3 Shown 8 lines Text

"GerardoM SarriaM" <gmsarria@tutopia.com> writes:

> On 10 May 2001, Pierre R. Mai wrote:
>
> > Hence generic-function-method-class will return
> the name and not the
> > class all over the place, and this could in
> theory lead to other
> > errors, beside the one Gerardo encountered.
>
> In my case, when I used the defmethod you wrote,
> in OpenMusic, there was a
> infinite loop that ends with a segmentation
> fault.

Hmmm, on the face of it this appears to be an unrelated problem,
IMHO.

> > If someone really needs this fixed in his 18c
> binary, I could produce
> > a file that does the correct fixes at
> load-time. Contact me in that
> > case.
>
> Please tell me the file to modify and what to
> modify in it. I mean, to compile Cmucl again. it
> does not matter to me to do it some times.

No recompilation necessary. Compile and load the appended patch file in your running CMU CL, and the problem should be fixed. Whether that will solve your infinite loop and segfault problem remains doubtful, though. At least the following test-case seems to work OK after the patch is applied, but then again it seemed to work OK after applying my defmethod:

```
(defclass themethod (pcl::standard-method) ((blub :initform 5)))

(defgeneric foo (x y) (:method-class themethod))

(defmethod foo ((x integer) (y integer)) (* x y 42))
```

Now (foo 1 2) will return 84.

This works in compiled files as well.

Regs, Pierre.

[Part 2, Application/OCTET-STREAM 3.8KB.]
[Cannot display this part. Press "V" then "S" to save in a file.]

```
    [ Part 3: "Attached Text" ]



--
Pierre R. Mai <pmai@acm.org>                    http://www.pmsf.de/pmai/
 The most likely way for the world to be destroyed, most experts agree,
 is by accident. That's where we come in; we're computer professionals.
 We cause accidents.                            -- Nathaniel Borenstein
```

Anexo D. General Public Licence - GPL

Anexo D. General Public Licence - GPL

La GNU General Public Licence (GPL) es un tipo de licencia creada por la Free Software Foundation para garantizar la libertad de compartir y distribuir copias del software gratis y su código fuente. Esto permite poder modificar el software o usar parte de él en nuevos programas.

Para proteger el software, se otorga un *copyright* y una licencia que ofrece un permiso legal de copiar, distribuir y/o modificar el software. Sin embargo, no se ofrece ninguna garantía, lo que ocaciona que si una persona modifica el software y lo distribuye, se debe entender que éste nuevo software no es el original y puede no reflejar la idea del autor del software original. La GPL no permite incorporar el programa a un software con licencia propietaria.

Si se desarrolla un nuevo programa y se desea cobijarlo bajo los términos de la GPL, se debe adicionar una copia de la licencia y escribir al principio de cada archivo del código fuente el *copyright* y el convienio de exclusión de garantía del software de la siguiente forma:

una línea para el nombre del programa y una idea de lo que hace
Copyright (C) *año nombre del autor*

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software

Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

E información adicional de cómo contactar al author vía correo electrónico o convencional